PROGETTO 2 - Distributed Systems and Big Data

UniCT - Anno Accademico 2025/2026

Gioele Messina - Luca Camiolo

1. DESCRIZIONE APPLICAZIONE

Il progetto estende l'architettura sviluppata nell'homework precedente, introducendo il pattern di resilienza circuit breaker, un API gateway come singolo punto di ingresso al sistema, ed un message broker Kafka, che consente, dopo aver scaricato le informazioni sugli aeroporti di interesse degli utenti tramite il servizio API di Open Sky, di inviare automaticamente una notifica via posta elettronica agli utenti ogni qualvolta il numero di voli in arrivo e/o in partenza per gli aeroporti di loro interesse superi o scenda al di sotto di certe soglie configurabili dagli utenti.

Circuit Breaker:

Il Circuit Breaker è un pattern che mira a migliorare la robustezza delle comunicazioni remote. Il problema principale delle comunicazioni remote è che possono fallire oppure rimanere bloccate, causando attese potenzialmente indefinite. Il Circuit Breaker serve proprio a prevenire queste situazioni, interrompendo temporaneamente le chiamate verso un servizio remoto non disponibile, controllando periodicamente se il sistema è tornato a funzionare, ed in caso positivo permetterà nuovamente di mandare richieste al sistema remoto. Il circuit breaker permette di risparmiare risorse, cioè, evita di fare richieste che non avranno risposta, migliora la user experience in quanto interrogando il circuit breaker possiamo vedere se è aperto o chiuso e quindi capire o meno se ci sono disservizi.

Il Circuit Breaker ha 3 stati fondamentali:

• Closed State: consente di mandare tutte le richieste al servizio remoto

• Open-State: le richieste non vengono mandare al servizio remoto in quanto non è disponibile.

• Half-Open-State: è uno stato in cui il Circuit Breaker si prende carico di capire quando il servizio torna disponibile.

Inizialmente il circuito è in stato di closed consentendo quindi di mandare le richieste al servizio remoto. A seguito di un certo numero di fallimenti consecutivi (configurabile) il circuito passa in stato Open non consentendo di inoltrare le richieste al servizio remoto.

Quando il circuito è nello stato open open, per verificare se il sistema remoto è tornato a funzionare, effettuerà periodicamente un controllo per verificare se il sistema funziona. Se il sistema sembra funzionare passerà, dopo un timeout, allo stato Half open. Lo stato half open durerà fino a quando o non si ha un fail, ed in tal caso si passerà nuovamente allo

stato Open, oppure fino a quando non si avranno un certo numero di successi consecutivi (configurabili).

API gateway:

L'API gateway è un pattern intermediario che si pone tra il client e i microservizi backend, nascondendo la complessità del sistema al client e fornendo un unico punto di ingresso. Grazie all'API gateway, il client non deve conoscere tutti i microservizi, i servizi offerti, gli indirizzi o le porte su cui sono in ascolto. Questo disaccoppia il client dall'architettura interna del sistema, semplificando l'interazione. Nel progetto, l'API gateway è stato implementato utilizzando NGINX, configurato come reverse proxy per smistare le richieste verso i corretti microservizi. NGINX gestisce quindi il routing delle richieste, la sicurezza tramite SSL e funge da unico punto di accesso per tutte le comunicazioni tra client e backend.

Message broker Kafka:

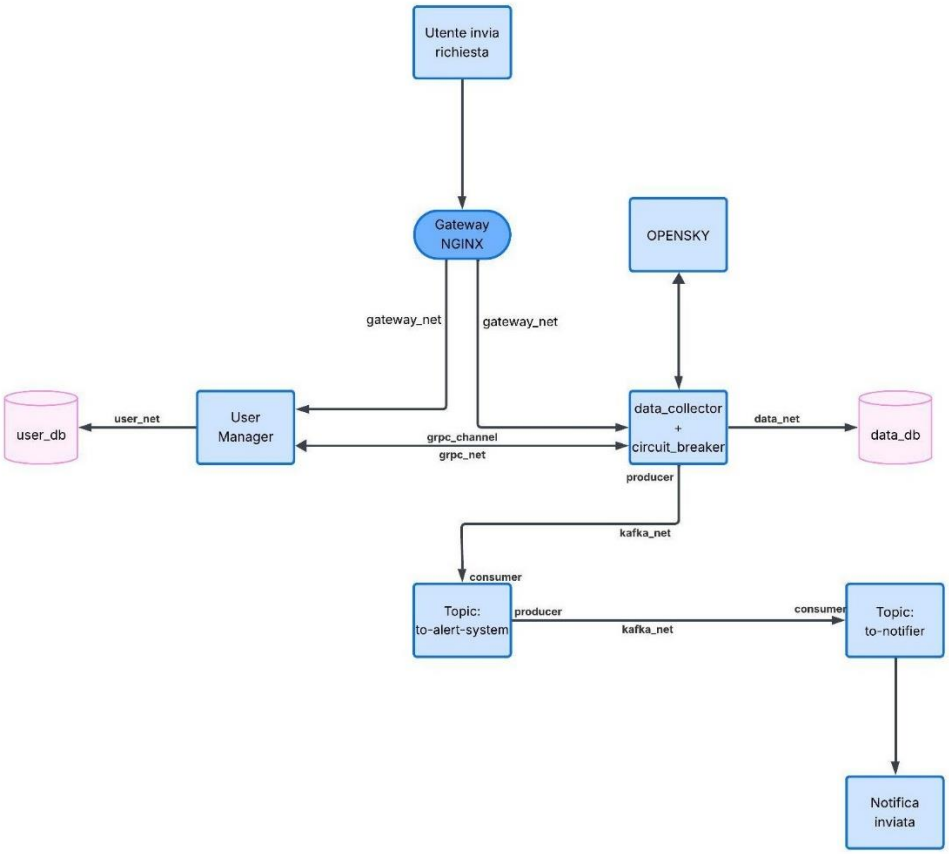
Il broker Kafka si occupa di creare e gestire due topic ***to-alert-system*** e ***to-notifier***, che fungono da canali di comunicazione asincrona tra i microservizi del sistema. In particolare, il data_collector dopo aver recuperato da OpenSky le ultime informazioni aggiornate degli aeroporti di interesse e aggiornato il database, agisce come producer inviando un messaggio sul topic Kafka to-alert-system andando a fornire all'interno di un payload le informazioni scaricate e l'elenco degli utenti che sono interessati a tali informazioni. Il consumer del topic *to-alert-system* elabora i messaggi andando a verificare, per ogni utente interessato a tali informazioni, se il numero delle informazioni scaricate super la soglia alta o se non supera la soglia bassa, in tal caso si comporterà da producer andando a produrre un messaggio che invierà sul topic to-notifier. Il consumer del topic to-notifier consuma i messaggi prodotti dall'alert systerm e, dopo averli elaborati si occuperà di inviare un'opportuna email all'utente interessato tramite il servizio SMTP (Gmail).

2. SCELTE PROGETTUALI

Per l'implementazione delle nuove funzionalità sono stati introdotti quattro nuovi container: un container per il broker Kafka (kafka), un container per il componente AlertSystem (alert_system), uno per il componente Notifier System (notfier_system) e l'ultimo per il gateway NGINX (apigw).

Sono state introdotte due nuove network Docker, denominate "***kafka_net***" e "***gateway_net***" con l'obiettivo di isolare le comunicazioni tra i diversi componenti del sistema. In particolare, la network gateway_net è stata introdotta per la comunicazione tra l'API Gateway NGINX e i microservizi User Manager e Data Collector, mentre la network kafka_net è dedicata per la comunicazione con il broker Kafka, garantendo un maggiore isolamento e sicurezza.

Per aumentare il livello di sicurezza, è stato scelto di utilizzare il protocollo **HTTPS** per le richieste indirizzate all'API Gateway NGINX. Una volta ricevute le richieste HTTPS, il gateway provvede a convertirle in richieste HTTP e a inoltrarle al microservizio di destinazione.



3. DETTAGLI IMPLEMENTATIVI

Circuit Breaker:

Il Circuit breaker è stato integrato come modulo nel Data Collector.

I parametri di configurazione del circuit breaker sono:

- failure_threshold: Questo rappresenta la soglia del numero di errori consecutivi, che una volta superata, il circuito passa allo stato OPEN. Se non viene esplicitamente fornito un valore verrà utilizzato il valore di default ossia 5, quindi dopo 5 fallimenti consecutivi il circuit breaker passa allo stato OPEN.
- timeout: rappresenta il tempo di attesa durante lo stato OPEN, allo scadere del quale il Circuit Breaker passa allo stato HALF-OPEN. In questo stato viene consentito il passaggio di una richiesta di test; se non vengono riscontrati problemi, il Circuit Breaker passa allo stato CLOSED, altrimenti ritorna allo stato OPEN.
- xpected_exception: indica il tipo di eccezione che verrà considerata come fallimento. Se non viene specificato alcun valore, tutte le eccezioni di tipo Exception saranno considerate come errori.

Il Circuit Breaker inizialmente si trova nello stato CLOSED, consentendo l'invio delle richieste verso il sistema remoto. Dopo ogni chiamata al sistema remoto, possono verificarsi due scenari:

1) Chiamata completata con successo:

   - Non viene sollevata alcuna eccezione, il che significa che tutto è andato correttamente.
   - In questo caso, il contatore di errori consecutivi (counter_failure) viene resettato, in modo da tracciare solo gli errori consecutivi.

2) Chiamata interrotta da un'eccezione expected_exception:
   - La chiamata viene interrotta e il contatore di errori (counter_failure) viene incrementato.
   - Viene registrato il timestamp dell'errore.
   - Se il contatore supera la soglia configurata (failure_threshold), lo stato del Circuit Breaker cambia in OPEN.

Quando il Circuit Breaker si trova nello stato OPEN e arriva una nuova richiesta da inoltrare al servizio remoto, si distinguono due casi:

1) ***time_since_failure=time.time()-self.last_failure_timestamp*** è minore del *timeout*, ciò significa che la richiesta non potrà passare e verrà sollevata un'eccezione che indica che il circuit breaker è in stato OPEN.
2) ***time_since_failure=time.time()-self.last_failure_timestamp*** è maggiore del *timeout*; quindi, la richiesta potrà essere inoltrata e lo stato del Circuit Breaker cambierà da OPEN in HALF OPEN.

   Se la chiamata va a buon fine:

- Il contatore di errori (counter_failure) viene azzerato
- Lo stato del Circuit Breaker diventa CLOSED.

  Se la chiamata solleva un'eccezione:

- Viene registrato il timestamp
- il contatore viene incrementato
- Circuit Breaker torna nello stato OPEN.

API gateway

l'API gateway è stato implementato utilizzando NGINX, in particolare è stato definito un container denominato "nginx_container", configurato per essere sempre in ascolto sulla porta 8443 in attesa di richieste https da smistare verso il data_collector o lo user_manager. NGINX agisce come reverse proxy inoltrando le richieste ai microservizi senza che l'utente debba conoscere l'indirizzo del microservizio.
La configurazione prevede due location per instradare le richieste che riceve dall'utente (ad es tramite postman all'url https://localhost:8443/) verso l'opportuno microservizio che si occuperà di eseguire la richiesta.
La location *location/user_manager/* intercetta ed instrada le richieste con prefisso */user_manager/* verso il microservizio User Manager. Ad es se la richiesta che riceve è

https://localhost:8443/user_manager/register, essendo che la richiesta matcha con *location/user_manager/* inoltrerà al microservizio user_manager la richiesta /register, in quanto il prefisso /user_manager verrà tolto da NGINX durante l'inoltro poiché il proxy pass,definito dentro la location, possiede lo "/" finale (proxy_pass http://user_manager/). La seconda location è *location/data_collector/*, il cui comportamento è analogo a quello della precedente location, con la differenza che le richieste che matchano verranno inoltrate verso il microservizio data_collector.

Broker Kafka

Per la gestione della messaggistica tra i microservizi è stato introdotto un broker Kafka, eseguito in un container dedicato denominato **broker_kafka**. Il broker si occupa di:

- **Creazione e gestione dei topic**: i messaggi vengono organizzati in topic, che fungono da canali logici di comunicazione. Nel progetto sono stati utilizzati i topic **to-alert-system** e **to-notifier**.
- **Persistenza dei messaggi**: Kafka memorizza i messaggi su disco nel percorso configurato (KAFKA_LOG_DIRS) garantendo durabilità dei dati.
- **Partizioni e repliche**: Nel setup attuale, i topic creati hanno 1 partizione e 1 replica, quindi non è presente ridondanza.
- **Gestione degli offset**: Kafka mantiene lo stato di lettura dei messaggi tramite il topic interno __consumer_offsets, permettendo ai consumer di sapere quale messaggio è già stato elaborato.
- **Configurazione dei listener**: il broker Kafka è esposto sia all'interno della rete Docker (kafka_net) sulla porta interna 9092, sia all'esterno tramite la porta mappata 29092 sull'host. Il listener PLAINTEXT indica che la comunicazione tra producer/consumer e broker avviene senza cifratura
- **Isolamento della comunicazione**: è stata creata una network dedicata kafka_net per isolare il traffico verso il broker e garantire maggiore sicurezza.
- **Healthcheck**: il broker verifica periodicamente la propria disponibilità tramite il comando kafka-broker-api-versions, per garantire che sia pronto a ricevere messaggi dai client.

Configurazione producer:

- **Data_collector:** Ogni volta che scarica informazioni sugli aeroporti di interesse agli utenti, produce un messaggio su Kafka sul topic to-alert-system contenente: l'icao dell'aeroporto, il numero di informazioni scaricate e l'elenco di tutti gli utenti che sono interessati a quelle informazioni.
    - **Parametri di configurazione:**
- **Alert System:** scrive sul topic to-notifier un messaggio contenente delle informazioni correlate all'utente da notificare tramite e-mail; ovvero: e-mail, subject ed il corpo che dovrà avere l'e-mail.
    - **Parametri di configurazione:** Per quanto riguarda i parametri di configurazione sono stati impostati i seguenti parametri per il producer alert_system

```
producer_conf = {
    'bootstrap.servers': KAFKA_BOOTSTRAP_SERVERS,
    'acks': 'all',
    'retries': 3,
}
```

ed i seguenti per il producer data_colector:

```
producer_conf = {
    'bootstrap.servers': KAFKA_BOOTSTRAP_SERVERS,
    'acks': 'all',
    'retries': 3,
     'linger.ms': 500,
     'batch.size': 16384,
    'max.in.flight.requests.per.connection': 1
}
```

- **bootstrap.servers:KAFKA_BOOTSTRAP_SERVERS** specifica l'indirizzo iniziale del broker Kafka utilizzato dai client per ottenere i metadata del cluster.
- **acks: 'all':** garantisce che il broker leader restituisca un ack solo dopo aver scritto il messaggio su tutte le repliche In-Sync (ISR). Questo approccio massimizza l'affidabilità: se il broker leader dovesse fallire, le repliche avranno comunque una copia consistente del messaggio. Tuttavia, questa configurazione può aumentare leggermente la latenza.
- **retries: 3**: tenterà di inviare il messaggio fino a tre volte prima di fallire definitivamente.
- **batch.size:16384**: rappresenta la dimensione massima del batch per partizione in byte. Appena il batch si riempie viene inviato immediatamente, se non viene riempito entra in gioco il parametro successivo.
- **linger.ms:**
  - impostato a 0 nell'alert_system: I messaggi sono inviati immediatamente (non inviamo batch ma singoli messaggi in quanto produce 1 solo messaggio alla volta).
  - Impostato a 500 nel data_collector, ossia aspetterà 500ms prima di inviare i messaggi al broker se il batch non viene riempito, se il batch si riempie prima di 10ms allora viene inviato immediatamente.
- **max.in.flight.requests.per.connection: 1** È un parametro che controlla il numero di messaggi che il producer può mandare a Kafka senza che prima sia stata ricevuta una risposta (ack) per un messaggio già inviato > parametro fondamentale per garantire l'ordine dei messaggi in una partizione, se fosse >1 potrebbe non essere rispettato l'ordine temporale dei messaggi.

Configurazione consumer:

- **Alert System:** effettua la sottoscrizione al topic to-alert-system. Ogni messaggio sul topic to-alert-system rappresenta una notifica che i dati nel database sono stati

aggiornati dal Data Collector. Il consumer, una volta recuperato il messaggio, si occupa di verificare per ciascun utente se il numero di informazioni scaricate dal Data Collector oltrepassano/stanno al di sotto delle soglie indicate dall'utente. Una volta processato il messaggio si occupa di fare il commit, comunicando al Broker che ha processato il messaggio.

- **Notifier System:** effettua la sottoscrizione al topic to-notifier: ogni messaggio contiene l'indirizzo e-mail ed il contenuto da inviare all'utente nel caso in cui si è verificata una certa condizione con una delle due soglie. Una volta inviata l'e-mail effettuerà il commit per comunicare che ha processato il messaggio.
  Per l'invio dell'e-mail è stato utilizzato il servizio SMTP di Gmail.
  - **Parametri di configurazione: :**Per quanto riguarda i parametri di configurazione sono stati impostati i seguenti parametri per entrambi i consumer (alert_system, notifier_system)

```
'bootstrap.servers': KAFKA_BOOTSTRAP_SERVERS,
'group.id': 'notifier_group',
'auto.offset.reset': 'earliest',
'enable.auto.commit': False
```

    - **bootstrap.servers:KAFKA_BOOTSTRAP_SERVERS** specifica l'indirizzo iniziale del broker Kafka utilizzato dai client per ottenere i metadata del cluster
    - **group.id: 'notifier_group'** definisce il gruppo al quale appartiene il consumer, questo permette di implementare il meccanismo dei competing consumer nel caso in cui si fossero usate più partizioni e fossero stati replicati i consumer.
    - **auto.offset.reset: 'earliest'** indica che, al momento della sottoscrizione a un topic, il consumer recupera i messaggi a partire dall'inizio della partizione.Questo comportamento viene applicato solo nel caso in cui il consumer non abbia offset precedentemente salvati per il gruppo di consumo. In alternativa, il valore latest consente al consumer di leggere solo i nuovi messaggi prodotti dopo la sottoscrizione, ignorando quelli precedenti.
    - **enable.auto.commit: False**, in questo modo, viene implementato il meccanismo di commit in modo customizzato, in particolare, in tutti i consumer si è scelto di committare solo successivamente all'elaborazione del messaggio in modo da prevenire il rischio di perdere messaggi (a scapito di eventuali duplicazioni in caso di riavvii del consumer). In questo modo il consumer comunica al broker Kafka di aver correttamente elaborato il messaggio solamente dopo che effettivamente questo è stato processato, richiedendo il salvataggio dell'offset associato all'ultimo record processato all'interno del topic.

4. MODIFICHE EFFETTUATE

4.1 DATA COLLECTOR E DATABASE

Per implementare il meccanismo di notifica tramite e-mail sono state introdotte all'interno della tabella user_interest nel data_db due soglie ***low_value*** e ***high_value.*** Tali soglie possono essere specificate dall'utente durante la fase di sottoscrizione di un nuovo interesse, se queste non vengono specificate assumeranno un valore di default. Se

l'utente desidera, è stata introdotta la possibilità, tramite la nuova route ***modifieTreshold*** nel Data_collector, di poter modificare per ciascun interesse uno od entrambi i valori delle soglie.

4.2 ESECUZIONE E TEST

Prima di eseguire l'applicazione, è necessario configurare correttamente le **variabili d'ambiente** richieste dai servizi esterni utilizzati, in particolare **OpenSky Network** e il servizio **SMTP di Gmail** per l'invio delle email.

Configurazione OpenSky Network

Per l'accesso alle API di **OpenSky Network**, è necessario aprire il file .env e sostituire i valori dei seguenti campi:

- OPENSKY_CLIENT_ID
- OPENSKY_CLIENT_SECRET

con le credenziali effettive fornite da OpenSky Network, ottenute a seguito della registrazione al servizio tramite il link ufficiale di OpenSky.

Configurazione SMTP Gmail con App Password

Per il servizio di invio email tramite **SMTP Gmail**, **non è possibile utilizzare la password principale dell'account Google**.
Google richiede l'utilizzo di una **App Password**, una password dedicata generata appositamente per applicazioni esterne.

Per utilizzare correttamente l'SMTP di Gmail è quindi necessario:

- Un **account Google**
- L'**autenticazione a due fattori (2FA)** attivata sull'account

Cliccando nel seguente link:
https://support.google.com/mail/answer/185833?hl=en

e navigando nell'apposita sezione "**Create and manage your app passwords**", sarà possibile ottenere la propria Password personale da utilizzare per testare correttamente l'invio delle email.

Questa password deve essere inserita nel file .env dell'applicazione al posto della password dell'account Gmail, ad esempio:

- GMAIL_USER → indirizzo Gmail
- GMAIL_APP_PASSWORD → App Password generata

Una volta completata la configurazione delle variabili d'ambiente, l'applicazione può essere avviata tramite **Docker Compose**, eseguendo il seguente comando da terminale:

docker-compose up –build

Test dell’applicazione

Nota sul file api_Test.http

Il file api_Test.http, utilizzato in precedenza per testare le API tramite chiamate HTTP dirette, **non è più presente**.
Questo perché, con l’introduzione del protocollo **HTTPS** e l’utilizzo di **SSL** per garantire una connessione sicura, le chiamate HTTP non risultano più utilizzabili direttamente da file .http come avveniva in precedenza.

Test tramite Postman

Per il testing delle API viene quindi utilizzato **Postman**, che consente di testare correttamente le chiamate HTTPS in modo semplice e strutturato.

Per testare il corretto funzionamento dell’applicazione è sufficiente seguire i seguenti passaggi:

1. Aprire **Postman**
2. Cliccare in alto a sinistra su **“Import”**
3. Selezionare i file .json (**UserManager** e **DataCollector**) presenti all’interno della cartella **Postman** del progetto
4. Eseguire le varie **chiamate** già predisposte per ciascuna delle API disponibili

In questo modo è possibile verificare il corretto comportamento di tutti i servizi esposti dall’applicazione in maniera sicura e controllata.