

## ***Progetto #2: Prova In Itinere - Relazione***

Venezia Vincenzo  
Corso di Ingegneria Informatica LM32 UNICT  
Data 19-12-2025

## **1. Introduzione al progetto**

Il presente lavoro si propone di estendere un sistema esistente per la raccolta e il monitoraggio dei dati relativi ai voli degli aeroporti, introducendo nuove funzionalità volte a migliorare l'affidabilità, la scalabilità e la tempestività delle notifiche agli utenti. L'obiettivo principale consiste nell'integrare meccanismi avanzati di comunicazione asincrona, oltre a ridefinire alcune componenti chiave dell'architettura del sistema.

In particolare, il progetto si concentra su quattro aspetti fondamentali:

### **1. Affidabilità delle comunicazioni con Open Sky Network**

Tutte le operazioni verso l'API esterna di Open Sky Network saranno protette mediante un meccanismo di *Circuit Breaker*, al fine di prevenire malfunzionamenti a cascata in caso di errori o latenza prolungata.

### **2. Notifiche asincrone agli utenti**

Verrà implementato un sistema di alert asincrono che permetta agli utenti di ricevere notifiche (ad esempio via email) ogni volta che il numero di voli in arrivo o in partenza da un aeroporto supera determinate soglie definite dall'utente.

### **3. Integrazione di un API Gateway e di un Message Broker**

L'architettura del sistema sarà aggiornata con l'introduzione di un API Gateway (NGINX) come punto di ingresso unificato e di un broker Kafka, utilizzato per gestire la comunicazione asincrona tra i diversi servizi, garantendo scalabilità e disaccoppiamento tra i componenti.

#### **4. Estensione delle funzionalità utente e del DataCollector**

Gli utenti potranno fornire parametri aggiuntivi, denominati *high-value* e *low-value*, per definire soglie personalizzate di alert. Il DataCollector, dopo aver aggiornato il database con i dati più recenti, invierà messaggi su Kafka verso il sistema di alert, che si occuperà di analizzare le soglie e notificare gli utenti in modo autonomo.

## 2. Implementazione Circuit Breaker

### 2.1. Introduzione al Pattern Circuit Breaker

Il **Circuit Breaker** è un pattern di resilienza utilizzato per prevenire il sovraccarico di un sistema quando un servizio esterno è in errore o non risponde. L'idea di base consiste nel monitorare le chiamate a un servizio esterno e, quando si rilevano errori consecutivi superiori a una soglia prestabilita, interrompere temporaneamente le richieste verso il servizio (stato **OPEN**). Dopo un periodo di recupero definito, il circuito passa in uno stato intermedio (**HALF\_OPEN**) per testare se il servizio è nuovamente disponibile. In caso di successo, il circuito ritorna allo stato **CLOSED**, permettendo nuovamente le chiamate.

Nel progetto, tutte le operazioni verso l'API esterna **Open Sky Network** sono protette tramite un Circuit Breaker. Questo approccio garantisce che eventuali malfunzionamenti o latenze eccessive dell'API non compromettano la stabilità complessiva del sistema.

L'implementazione del Circuit Breaker nel file `circuit_breaker.py` è basata sul codice fornito dai professori durante le lezioni. Le caratteristiche principali includono:

- **Soglia di errori (failure\_threshold)**: numero massimo di fallimenti consecutivi consentiti prima di aprire il circuito.
- **Timeout di recupero (recovery\_timeout)**: intervallo di tempo dopo il quale il circuito tenta di riaprire le chiamate verso il servizio.
- **Gestione delle eccezioni (expected\_exception)**: tipo di eccezioni che incrementano il contatore dei fallimenti.

- **Stati del circuito:** CLOSED (tutte le chiamate sono permesse), OPEN (nessuna chiamata è permessa) e HALF\_OPEN (prova di chiamata per verificare se il servizio è tornato disponibile).
- **Thread safety:** grazie all'uso di un lock, il Circuit Breaker può essere utilizzato in contesti multi-thread senza rischi di race condition.

## 2.2. Implementazione in Data Collector

Nel progetto, il **Circuit Breaker** viene utilizzato per proteggere tutte le chiamate verso l'API esterna **Open Sky Network**, evitando che eventuali malfunzionamenti o latenze eccessive compromettano l'intero sistema.

Il file `circuit_breaker.py` contiene la classe `CircuitBreaker`, basata sul codice fornito dalla docente durante le lezioni. La classe definisce tre stati principali: CLOSED, OPEN e HALF\_OPEN, e gestisce il conteggio dei fallimenti, il timeout di recupero e la gestione thread-safe delle chiamate.

Nel codice del Data Collector, le interazioni con l'API esterna vengono incapsulate tramite il metodo `call` del Circuit Breaker. In particolare:

### 1. Protezione delle chiamate all'API

Ogni richiesta verso Open Sky Network viene passata al metodo `call` del Circuit Breaker, che verifica lo stato corrente del circuito prima di effettuare la chiamata.

### 2. Gestione degli errori

- Se il circuito è OPEN, la chiamata viene immediatamente rifiutata e viene sollevata un'eccezione `CircuitBreakerOpenException`.

- Se la chiamata genera un'eccezione prevista (expected\_exception), il contatore dei fallimenti viene incrementato. Quando il numero di fallimenti consecutivi supera la soglia (failure\_threshold), lo stato del circuito passa a OPEN.
- Dopo il timeout di recupero (recovery\_timeout), il circuito entra nello stato HALF\_OPEN e permette una chiamata di prova per verificare se il servizio è nuovamente disponibile.

### 3. Integrazione con il Data Collector

Sebbene il codice principale del Data Collector che hai condiviso non mostri direttamente le chiamate a Open Sky Network, il flusso operativo previsto è che ogni funzione che interroga l'API esterna sia avvolta dal Circuit Breaker. Questo garantisce che, in caso di malfunzionamenti dell'API, il sistema continui a funzionare senza interruzioni, proteggendo il database e le notifiche agli utenti.

### 4. Benefici

L'uso del Circuit Breaker assicura:

- **Resilienza:** riduce il rischio di errori a cascata dovuti a servizi esterni non disponibili.
- **Stabilità:** mantiene operativo il Data Collector anche in condizioni di errore temporaneo.
- **Controllo dei tempi di attesa:** evita che richieste lunghe o fallite blocchino il processo di raccolta dati.

## 3. API Gateway

### 3.1. API Gateway NGINX

Nel progetto, è stato introdotto un **API Gateway** basato su **NGINX**, che funge da punto di ingresso unificato per tutte le richieste provenienti dagli utenti verso i diversi servizi del sistema. L'obiettivo principale dell'API Gateway è:

- **Centralizzare le richieste** verso i microservizi, evitando che i client debbano conoscere direttamente gli indirizzi interni dei container.
- **Gestire il routing** verso i servizi appropriati, come User Manager e Data Collector.
- **Fornire un punto unico per monitoraggio e scalabilità**, semplificando l'aggiunta futura di logica di sicurezza, autenticazione o caching.

### 3.2. Configurazione NGINX

La configurazione di NGINX, definita in **nginx.conf**, specifica le location per ogni endpoint dei microservizi. Alcuni esempi:

- /register, /user/, /health/usermanager: instradati verso il servizio **User Manager**.
- /add\_airport, /remove\_airport, /airports/, /last\_flight/, /avg/, /health/datacollector: instradati verso il servizio **Data Collector**.

Grazie a questa configurazione, tutte le richieste HTTP passano prima dall'API Gateway, che le inoltra al servizio corretto, semplificando la gestione degli indirizzi e delle porte interne.

### 3.3. Implementazione in docker-compose.yaml

```
services:

# ----- API GATEWAY -----
api_gateway:
  image: nginx:latest
  container_name: api_gateway
  ports:
    - "8080:80"
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf
  depends_on:
    - user_manager
    - data_collector
  networks:
    - app-network
```

## 4. Implementazione Broker Kafka

### 4.1. Introduzione a Kafka e ai nuovi microservizi

Per estendere il sistema con funzionalità di **notifica asincrona** agli utenti, è stato introdotto un **message broker Kafka**, che permette la comunicazione disaccoppiata tra i diversi servizi. Kafka consente di inviare messaggi tra microservizi senza che essi siano direttamente connessi tra loro.

### 4.2. Implementazione “producer Kafka” in DataCollector

Il **Data Collector** è stato aggiornato per integrare Kafka nel flusso operativo:

1. Dopo aver recuperato i dati sui voli per gli aeroporti di interesse e aggiornato il database, il Data Collector invia un messaggio su un **topic Kafka dedicato** (to-alert-system).
2. Il messaggio contiene le informazioni aggiornate sugli aeroporti, tra cui il numero di voli in arrivo e in partenza per ogni aeroporto monitorato.
3. Questa modifica rende il Data Collector un **producer Kafka**, delegando ad altri servizi l’elaborazione delle soglie di alert, invece di gestirle internamente.

L’integrazione di Kafka avviene tramite il modulo `kafka_producer.py`, nel quale viene istanziato un **KafkaProducer** configurato per comunicare con il broker Kafka presente nel sistema:

```

from kafka import KafkaProducer
import json

producer = KafkaProducer(
    bootstrap_servers="kafka:9092",
    value_serializer=lambda v: json.dumps(v).encode("utf-8")
)

```

Il producer utilizza:

- Il broker Kafka accessibile tramite il nome del servizio Docker (kafka:9092);
- Un **serializer JSON**, che consente di inviare messaggi strutturati e facilmente interpretabili dai consumer.

La funzione **notify\_alert\_system** incapsula la logica di invio del messaggio:

```

def notify_alert_system(email, airport_code, total_flights):
    """
    Invia un messaggio a Kafka topic 'to-alert-system'
    """
    msg = [
        "email": email,
        "airport_code": airport_code,
        "total_flights": total_flights
    ]
    producer.send("to-alert-system", msg)

```

Nel codice principale del **Data Collector**, la funzione `notify_alert_system` viene invocata in punti strategici del flusso applicativo. Un esempio significativo è l'endpoint `/add_airport`: `notify_alert_system(email, airport_code, total_flights)`

Dopo aver:

1. validato i parametri forniti dall'utente;
2. verificato l'esistenza dell'utente tramite il servizio User Manager;
3. aggiornato il database con i nuovi parametri (high\_value, low\_value);

il Data Collector invia un messaggio a Kafka per notificare che i dati relativi all'aeroporto sono stati aggiornati.

### 4.3. Microservizio AlertSystem

**Alert System** è un servizio indipendente che funge da **consumer Kafka** sul topic to-alert-system. Le sue responsabilità principali sono:

- Ricevere i dati aggiornati dal Data Collector.
- Per ogni profilo utente, confrontare il numero di voli con le soglie definite (high-value e low-value).
- Se viene superata una soglia, generare un messaggio di notifica con i parametri <utente, aeroporto, condizione di superamento soglia> e pubblicarlo su un altro topic Kafka (to-notifier).

Il microservizio AlertSystem agisce come **Kafka consumer** sul topic to-alert-system.

All'avvio del servizio, viene creato un KafkaConsumer configurato per leggere i messaggi dal broker Kafka:

```

consumer = KafkaConsumer([
    "to-alert-system",
    bootstrap_servers=bootstrap_servers,
    value_deserializer=lambda v: json.loads(v.decode("utf-8")),
    auto_offset_reset='earliest',
    group_id='alert-system-group'
])

```

Una volta connesso a Kafka, l'AlertSystem entra in un ciclo di ascolto continuo:

*for msg in consumer:*

```
    data = msg.value
```

Ogni messaggio contiene:

- airport\_code: codice dell'aeroporto;
- total\_flights: numero totale di voli rilevati.

Il servizio verifica la correttezza del messaggio e, in caso di dati validi, procede con le seguenti operazioni:

**1. Recupero dei profili utente:**

Tramite l'accesso al database, l'AlertSystem recupera tutti i profili associati all'aeroporto: *profiles = get\_profiles\_for\_airport(airport)*

**2. Verifica delle soglie:** Per ogni profilo, vengono estratti i valori

*high\_value* e *low\_value* e viene invocata la funzione:

```
condition = check_threshold(total, high, low)
```

**3. Produzione dell'evento di alert**

Se una soglia viene superata, l'AlertSystem invia un nuovo messaggio Kafka sul topic *to-notifier*:

```
notify_user(email, airport, condition)
```

In questo modo, l'AlertSystem assume anche il ruolo di **producer Kafka**, generando eventi di alert destinati al servizio di notifica.

#### 4.4. Microservizio AlertNotifier

Il **Alert Notifier** è un servizio indipendente che funge da **consumer Kafka** sul topic to-notifier. Le sue responsabilità includono:

- Ricevere i messaggi di alert generati dall'Alert System.
- Inviare notifiche all'utente finale tramite email o altri canali configurabili (ad esempio Telegram), includendo informazioni sul superamento della soglia (high-value o low-value).
- Gestire il formato dei messaggi (to, subject, body) e la consegna asincrona, senza bloccare altri componenti del sistema.

All'avvio del servizio, viene istanziato un KafkaConsumer configurato per leggere i messaggi di notifica:

```
consumer = KafkaConsumer(  
    TOPIC,  
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,  
    value_deserializer=lambda v: json.loads(v.decode("utf-8")),  
    auto_offset_reset="earliest",  
    enable_auto_commit=True,  
)
```

Ogni messaggio consumato dal topic to-notifier contiene le informazioni necessarie per inviare una notifica all'utente:

- indirizzo email dell'utente;

- aeroporto di riferimento;
- descrizione della condizione di superamento soglia.

Una volta ricevuto il messaggio, il servizio utilizza il modulo SMTP per inviare una email:

```
import smtplib
from email.message import EmailMessage

def send_email(to, subject, body):
    msg = EmailMessage()
    msg["From"] = "alerts@dsbd.local"
    msg["To"] = to
    msg["Subject"] = subject
    msg.set_content(body)

    # Configurare il tuo SMTP server
    with smtplib.SMTP("smtp", 25) as s:
        s.send_message(msg)
```

Il contenuto della mail è strutturato come segue:

- **To:** email dell'utente;
- **Subject:** codice dell'aeroporto;
- **Body:** descrizione della soglia superata, specificando se si tratta di soglia superiore (*high-value*) o inferiore (*low-value*).