Sistemi Distribuiti e Cloud Computing

Progetto B3: App a microservizi a scelta

Pieraldo Santurro
Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
Roma, Italia
santurro.pieraldo99@gmail.com

***Abstract* — Questa relazione descrive lo sviluppo di un sistema distribuito basato su microservizi per la gestione di biglietti per eventi.**

I. Introduzione

Il progetto realizzato consiste in un'applicazione con architettura a microservizi che fornisce una piattaforma per la gestione di biglietti per eventi con l'aggiunta di un'interfaccia web. Gli utenti sono di due tipologie: "user" e "admin", ed hanno la possibilità di:

- Registrarsi in modalità user o admin
- Effettuare il login
- Visualizzare gli eventi disponibili
- Aggiungere e rimuovere eventi (solo admin)
- Acquistare biglietti per eventi

L'interfaccia web è sviluppata in *Flask*, con funzionalità di autenticazione e gestione eventi completamente integrate via *gRPC*.

II. Architettura

L'architettura è composta da quattro microservizi di cui tre di tipo stateful che mantengono informazioni su database esterni, ed un microservizio che implementa l'interfaccia web. L'applicazione è stata sviluppata in linguaggio Go ed utilizza sia chiamate sincrone via *gRPC* che asincrone via *RabbitMQ*. Ogni microservizio è stato istanziato all'interno di un container *Docker*, permettendo flessibilità e scalabiltà, testata anche mediante le funzionalità di *Docker Swarm*, che permette l'istanziazione di più repliche dello stesso container.

Per l'architettura sono stati implementati i pattern *Database-per-service* e *Saga*, mentre per la tolleranza ai guasti sono stati implementati *Circuit-breaker* e meccanismi di *Retry-Timeout*.

III. Microservizi

A. User-Service

Il microservizio *User-Service* permette di effettuare *Login* e *Registrazione.* Nel caso di prima registrazione sarà possibile scegliere un nome utente, una password e scegliere il ruolo dell'account. Utilizza un'istanza *MongoDB* per il mantenimento delle informazioni sugli utenti. I servizi esposti sono:

- *Register*
- *Login*

B. Authentication-Service

Dedicato alla generazione e validazione di *Token JWT* contenenti ID utente, ruolo e username. Permette di interfacciarsi con gli altri microservizi per funzionalità di sicurezza sviluppabili in futuro. I serivizi esposti sono:

- *GenerateToken*
- *ValidateToken*

C. Ticket-Service

Microservizio dedicato alla gestione degli eventi. Un utente registrato come "user" sarà in grado solamente di effettuare acquisti dei biglietti presenti. Un utente registrato come "admin" sarà in grado di eseguire le stesse funzionalità di un utente "user" che di creare ed eliminare eventi. Viene fatto utilizzo di un'istanza MongoDB per il mantenimento delle informazioni. Comunica tramite RabbitMQ con il microservizio *Payment-Service* nel caso di acquisto dei biglietti. I servizi esposti sono:

- *ListEvents*
- AddEvent
- DeleteEvent
- *PurchaseTicket*

D. Payment-Service

Microservizio dedicato alla simulazione di pagamenti. Salva le informazioni su MongoDB, comunicando con *Ticket-Service* tramite RabbitMQ. I servizi esposti sono:

- *ProcessPayment*
- *StartConsumingTicketReservedEvent*

E. Web-Ui

È il *frontend* della piattaforma e funziona sostanzialmente come interfaccia *HTML* ai servizi, chiamando le funzioni esposte tramite *gRPCurl*.

IV. Design Pattern

A. Database-per-service

Per gestire lo stato legato ai diversi microservizi, è stato adottato il pattern *Database-per-service*. Tale pattern prevede l'utilizzo di un database privato per ogni microservizio che lo necessiti. Ciascun database è accessibile soltanto tramite la sua API da parte del microservizio che lo utilizza. Per ogni microservizio che ne fa utilizzo, sono state istanziate immagini di *MongoDB*.

B. Saga

Viene implementato un pattern Saga di tipo *Coreography-based* per gestire transazioni distribuite senza l'uso di un coordinatore centrale, garantendo consistenza eventuale tra i microservizi, alta scalabilità e disaccoppiamento. I microservizi coinvolti nel pattern Saga sono *Ticket-Service e Payment-Service* che comunicano tramite *RabbitMQ*, che utilizza due code code: *ticket-reserved-queue* e *payment-events-queue*. Il flusso della Saga consiste nella prenotazione di un biglietto tramite le API offerte da *Ticket-Service*, che creerà un messaggio *TicketReservedEvent* pubblicandolo su RabbitMQ nella coda *ticket-reserved-queue.* Il microservizio *Payment-Service* è subscriber della coda, legge il messaggio, lo consuma registrando il pagamento nel suo database MongoDB e crea un messaggio *Payment-Event* che pubblicherà sulla *payment-events-queue.* A questo punto l'acquisto potrebbe essere confermato o rifiutato da un servizio esterno. Ogni evento ha un *EventIstanceID* univoco per deduplicazione, permettendo quindi di evitare doppie elaborazioni nel caso di fallimenti.

C. Circuit-Breaker

Implementato per la tolleranza ai guasti nei microservizi. Non sono state utilizzate librerie già esistenti ma si è preferito creare da zero un codice in grado di garantire un servizio di Circuit-Breaking utile al nostro contesto. In particolare, il meccanismo è stato implementato per garantire tolleranza ai guasti ai microservizi che comunicano tramite *RabbitMQ*, per evitare la pubblicazione continua di messaggi. Nel caso di stato *closed*, i messaggi possono essere pubblicati su *RabbitMQ* senza problemi. Nel caso di più tentativi falliti, lo stato passa ad *open* e le chiamate successive non provano nemmeno a riconnettersi a *RabbitMQ*, evitando che il sistema si appesantisca. Il sistema è in uno stato *Half-Open* quando dopo un intervallo di tempo di 10 secondi, implementato tramite un meccanismo periodico, il circuito prova a inviare un messaggio: se la pubblicazione ha successo, il sistema torna *closed*, se fallisce, rimane in *open*.

V. Piattaforma software

A. Piattaforma di Sviluppo

Il progetto è stato eseguito tramite macchina virtuale WSL2 su un calcolatore che possiede le seguenti caratteristiche:

- Processore: 12th Gen Intel(R) Core (TM) i5-12500H 2.50 GHz
- RAM: 16,0 GB (15,7 GB utilizzabile)
- OS: Windows 11 Home – 23H2 • WSL2: Ubuntu 22.04.3 LTS

B. Docker

Per lo sviluppo di tale applicazione è stato utilizzato Docker, ovvero una piattaforma software che consente la virtualizzazione a livello di sistema operativo e che permette di creare, testare e distribuire applicazioni con rapidità. L'utilizzo che ne è stato fatto è quello della creazione di immagini per i container. È stato utilizzato anche *Docker-Swarm* per l'istanziazione e il test dell'applicazione in uno scenario dove ci sono più repliche attive.

C. gRPC

gRPC è un moderno sistema di remote-procedure-call sviluppato da Google. All'interno della nostra applicazione è utilizzato per la comunicazione sincrona tra microservizi.

D. RabbitMQ

RabbitMQ è un middleware di messaggistica orientato agli eventi che implementa il protocollo *Advanced Message Queuing Protocol (AMQP)*. All'interno della presente architettura a microservizi, *RabbitMQ* svolge un ruolo fondamentale per il coordinamento asincrono tra i servizi, in particolare nella gestione delle transazioni distribuite tramite pattern *Saga (Choreography-Based).* Le code principali configurate in *RabbitMQ* per questo progetto sono:

- *ticket-reserved-queue*: utilizzata dal microservizio *ticket-service* per inviare un evento di prenotazione biglietti. Questo evento viene successivamente consumato da *payment-service* per avviare l'elaborazione del pagamento.
- *payment-events-queue*: gestita *da payment-service*, consente di notificare l'esito del pagamento (positivo o negativo) agli eventuali servizi interessati (es. sistemi di logging o notifiche).

Entrambe le code vengono dichiarate con il flag di durabilità, in modo da garantire la persistenza dei messaggi anche in caso di riavvio del broker. Inoltre, viene mantenuta la semantica "*at-least-once delivery*", propria del protocollo *AMQP*: ciò significa che, se il messaggio non viene correttamente consumato e confermato, *RabbitMQ* provvede al reinvio automatico, garantendo l'affidabilità del sistema.
Questo approccio consente ai microservizi di rimanere fortemente disaccoppiati, facilitando la scalabilità orizzontale e migliorando la tolleranza ai guasti. Inoltre, in caso di temporanea indisponibilità del broker o dei consumatori, i messaggi vengono accodati fino al ripristino, mantenendo l'integrità dei flussi evento-driven tra i servizi.

VI. Librerie utilizzate

- *grpcurl*: Tool di testing CLI utilizzata per l'invocazione delle funzioni gRPC.
- *amqp091-go*: Libreria per la comunicazione in RabbitMQ
- *uuid:* Libreria utilizzata per la creazione di UUID per i ticket ed i pagamenti.

- *JWT:* Libreria utilizzata per generazione e validazione dei token nel servizio *Authentication-Service*
- *Flask*: Framework per la creazione e la gestione della *web-ui*.

VII. Sviluppi Futuri

L'applicazione, data la struttura del progetto, presenta molti spunti di miglioramento e implementazione aggiuntivi:

- Integrazione delle funzionalità di autenticazione di *Authentication-Service* per funzioni di sicurezza legate agli altri microservizi. Ad esempio, una richiesta di autenticazione di Token JWT nei servizi di pagamento o nella prenotazione dei biglietti
- Integrazione di servizi aggiuntivi per il pagamento dei ticket, ad esempio l'utilizzo di API esterne, oppure un servizio di notifica che comunichi con *User-Service* per l'invio di un'e-mail di conferma una volta avvenuto il pagamento.