SEAt

Progetto A2: Un'Applicazione Basata su Microservizi (Sistemi Distribuiti e Cloud Computing – A.A 2021/22)

Ludovico Zarrelli
Computer Science
Università di Roma "Tor Vergata"
Matricola 0316448
ludovico.zarr@gmail.com

Elisa Venditti
Computer Science
Università di Roma "Tor Vergata"
Matricola 0315909
elisavenditti99@gmail.com

***Abstract*— Il documento vuole presentare i punti salienti per la realizzazione di un'applicazione con architettura a micro-servizi.**

I. INTRODUZIONE

L'applicazione realizzata consiste in una piattaforma web per la gestione delle prenotazioni in un lido. I clienti possono selezionare uno dei tanti preventivi che un algoritmo di "*suggestion"* calcola per loro. Quest'algoritmo elimina le richieste che non rispondono a requisiti specifici per l'utente; inoltre, si occupa di ordinare i risultati della ricerca in base a criteri generali (come il voto medio dei lidi). Il cliente può gestire una o più carte di pagamento ed eventualmente effettuare il pagamento online. È disponibile una sezione in cui i clienti possono lasciare delle recensioni per i lidi ed esprimere una votazione (da 1 a 5 stelle) che viene utilizzata per ricavare il voto medio per lo stabilimento balneare. Inoltre, l'applicazione offre l'analisi del sentimento dei visitatori di una determinata città, in modo da aiutare gli utenti nella scelta del luogo di villeggiatura.

II. TECNOLOGIE

Fase di sviluppo:

- **Python** – linguaggio per lo sviluppo del codice dei micro-servizi;
- **HTML** e **Javascript** – interfaccia grafica;
- **gRPC** – framework open-source per le chiamate a procedura remota;
- **RabbitMQ** – garantisce una comunicazione asincrona tra i micro-servizi. Per renderlo compatibile con Python abbiamo utilizzato la libreria **Pika** (che implementa il protocollo AMQP);
- **Flask** – framework per la realizzazione dell'applicazione web;
- **SQLite** – SQL database engine (utilizzato dal micro-servizio stateful e dal service registry);
- **Boto3** – SDK Python per AWS;
- **AWS DynamoDB** – database NoSQL (utilizzato da tutti i micro-servizi stateless);
- **AWS SQS** – servizio che offre la gestione di code di messaggi (utilizzato nell'interazione con il service registry);
- **AWS Comprehend** - servizio di elaborazione del linguaggio naturale (NLP) che adotta il machine learning per svelare informazioni e collegamenti in un testo;
- **Circuitbreaker** – libreria Python per il supporto alle funzionalità di circuit breaker;
- **Geopy** – client Python utilizzato per ricavare la latitudine e la longitudine di una località;
- **Haversine** – libreria Python utilizzata per calcolare la distanza tra due punti nello spazio.

Nonostante avessimo già le code gestite con RabbitMQ, abbiamo inserito SQS in modo da sperimentare più tecnologie.

Fase di deploy:

- **Docker** – software per il supporto ai container;
- **Docker Compose** – orchestratore per container residenti su una singola macchina;
- **AWS EC2** – servizio Amazon che abbiamo utilizzato per fare il deploy dell'applicazione su istanze remote raggiungibili con un indirizzo IP pubblico;
- **Terraform** - tool software IaC open-source che ci consente di creare, cambiare e migliorare l'infrastruttura EC2 attraverso la dichiarazione di risorse AWS;
- **Ansible** - software open-source che ci consente di automatizzare la configurazione e l'esecuzione di task sulle le istanze EC2.

III. SCELTE ARCHITETTURALI

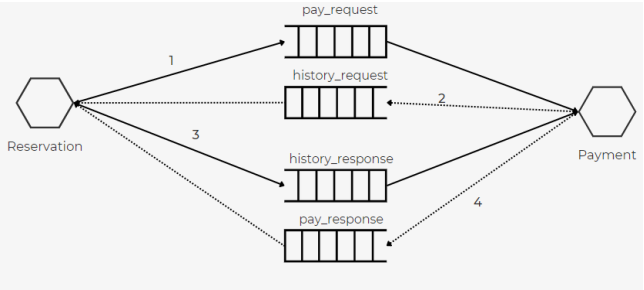
L'applicazione è pensata per risiedere in una sola macchina che ospita i vari container. In generale i servizi sono stateless e non vengono replicati ulteriormente. Questi servizi contattano un database comune (esterno alla macchina) e ognuno ha la responsabilità di accedere a determinate tabelle. Il microservizio stateful, invece, è stato replicato. Seguendo la logica delle capacità di business abbiamo ricavato i seguenti 8 microservizi: *frontend*, *serviceRegistry*, *accounting*, *reservation*, *email*, *quote*, *review* e *payment*. L'unico punto di accesso è il *frontend* che svolge le funzionalità di api gateway. Tale gateway è anche responsabile di bilanciare le richieste tra le istanze del servizio stateful (load balancing) seguendo una

politica "*round-robin"*. I container comunicano in modo sincrono tramite chiamate RPC e interagiscono seguendo l'approccio decentralizzato della coreografia. In questo modo evitiamo di concentrare i messaggi verso l'orchestratore centralizzato. Il micro-servizio *email* viene contattato in modo asincrono tramite delle code di messaggi: questo permette di disaccoppiare temporalmente la spedizione dell'email con le altre operazioni che vengono fatte dall'utente (che, ovviamente, non necessitano della spedizione dell'email per poter proseguire).

IV. PATTERNS

A. SAGA

Il pattern SAGA è stato applicato in 2 punti specifici dell'applicazione: prima per eliminare un account e i dati associati; poi per gestire la transazione di prenotazione e pagamento. Per rimanere coerenti con l'interazione basata su coreografia, non abbiamo implementato SAGA con un message broker centralizzato. Al contrario, i micro-servizi coinvolti, pubblicano degli eventi su code di messaggi apposite e cooperano attivamente per terminare la transazione. Analizzeremo solo la transazione per la prenotazione in quanto è la più complessa. In questo scenario abbiamo 2 micro-servizi che vengono coinvolti in una transazione a 3 step: prenotazione, pagamento e aggiornamento della storia utente. La storia utente contiene le informazioni delle prenotazioni passate (necessarie all'algoritmo di *suggestion* per ricavare le proposte adatte all'utente). Il passo 3 è stato inserito solo dopo il pagamento in quanto vogliamo che venga eseguito quando si è totalmente sicuri dell'esito positivo delle operazioni precedenti (vogliamo evitare traffico inutile verso il database esterno). Il flusso logico è descritto nella seguente figura.



Il servizio *reservation*, dopo aver aggiunto la prenotazione al database, si occupa di iniziare SAGA andando a pubblicare in pay_request. Il *payment* effettua il trasferimento del credito dalla carta del cliente a quella del lido coinvolto. Se l'operazione non va a buon fine pubblica direttamente una failure in pay_response. Altrimenti procede con SAGA andando a pubblicare in history_request. La richiesta viene recepita dal servizio *reservation* che aggiorna la tabella nel database esterno con i dati della prenotazione. Qualunque sia l'esito pubblica in history_response. Sarà il *payment*, dopo aver valutato se fare o no il rollback, a pubblicare il messaggio definitivo nella pay_response. È importante notare che ogni messaggio pubblicato nelle code porta con sé le informazioni legate alla transazione (come utente, e-mail, costo, budget …) in modo da consentire all'altro micro-servizio di svolgere le varie attività.

Nonostante l'asincronia introdotta dalle code, abbiamo dovuto rendere il meccanismo sincrono andando ad utilizzare delle procedure bloccanti. Questo perché la chiamata originale era una RPC sincrona che attendeva l'esito della transazione.

Come anticipato in precedenza, abbiamo utilizzato SAGA anche per l'eliminazione dell'account. Ci siamo assicurati di aver cancellato i dati relativi ai pagamenti (più sensibili), per poi passare alla rimozione dei dati nel database esterno.

Per garantire l'atomicità di ogni sequenza di operazioni (locale al micro-servizio) abbiamo utilizzato le transazioni offerte da DynamoDB. Per dettagli consultare la sezione V. Non abbiamo considerato la possibilità che il rollback di una di queste transazioni locali possa fallire. Si sarebbe innescato un meccanismo iterativo di difficile gestione.

B. CIRCUIT BREAKER

Il circuit breaker è il pattern architetturale che consente di mediare la reale chiamata di un servizio ed evitare che esso venga sovraccaricato in caso di fallimenti. Abbiamo scelto di applicare il pattern per le chiamate destinate al micro-servizio *Reservation*: quest'ultimo è il servizio *core* dell'applicazione perché contiene la logica di suggestion e di prenotazione (è colui che inizia SAGA). Nello specifico è l'api gateway a svolgere le funzionalità di un circuit breaker: (1) definisce una soglia di fallimenti consentiti prima di aprire il circuito; (2) predispone una callback da eseguire quando il circuito è aperto che ritorna un errore senza contattare il micro-servizio; (3) definisce un tempo in cui il circuito deve rimanere aperto. Per implementare tutte queste attività ci siamo serviti di una libreria di Python (vedi sezione II).

C. SERVICE-SIDE SERVICE DISCOVERY

Il server-side discovery è un pattern molto utilizzato nelle moderne applicazioni a micro-servizi che eseguono in un containerized environment dove il numero di istanze e la loro posizione in rete cambia dinamicamente.

Schema d'interazione:
Prevede la presenza di un *service registry* che, attraverso le notifiche dei micro-servizi, acquisisce informazioni sul loro indirizzo IP, porta, nome del servizio e hostname. Il registry memorizza le informazioni in un database locale.
L'*api-gateway* conosce la sua locazione in rete e, una volta istanziatosi, contatta il *service registry* per scoprire la locazione del micro-servizio d'interesse. Per modularizzare il codice, il gateway è stato suddiviso in diverse classi, ognuna con le sue responsabilità. In particolare, la classe adibita alla comunicazione con il micro-servizio X avrà il compito di scoprire la sua posizione per poter iniziare a dialogare.
Per migliorare le performance, la risposta che il *service registry* fornisce all'*api-gateway* viene memorizzata localmente in un attributo di classe per eventuali riutilizzi futuri. Questa è una soluzione ad-hoc per il nostro sistema: in teoria, il pattern prevederebbe la comunicazione con il *registry* ogni richiesta dal client. Nel nostro caso non è necessario poiché la porta e l'IP del container non cambiano dinamicamente. La porta è definita staticamente nel file di docker compose mentre l'IP/hostname è assegnato dal DNS di default. L'IP può cambiare ad ogni invocazione del comando docker-compose up che crea e avvia i containers.

Per realizzare il pattern abbiamo utilizzato due diversi modelli di comunicazione:

- Una coda di messaggi (gestita da AWS SQS) sulla quale il *registry* riceve dai servizi le informazioni sulla loro locazione in rete. A tal proposito è utilizzato un meccanismo di pooling per acquisire i messaggi da tutti. Dopo 20 secondi, il *registry* si mette in ascolto per eventuali chiamate gRPC.
  In questo caso, stiamo garantendo disaccoppiamento spaziale, temporale e di sincronia lato client (il servizio non è bloccato in attesa di una risposta).
- Un middleware di comunicazione RPC (attraverso la piattaforma open-source gRPC) tra il *gateway* e il *registry*. Questa scelta è dettata dalla necessità di avere una comunicazione transiente e "response-based synchronous". Sincrona poiché il client deve essere bloccato in attesa della risposta dal *registry*; transiente poiché se la consegna non è disponibile viene scartato.

Il *service registry* costituisce un unico punto di rottura per la nostra applicazione. Perciò abbiamo deciso di inserire un meccanismo di protezione: nel caso in cui il *registry* non sia disponibile o contattabile, il *gateway* utilizza un binding statico per contattare gli altri micro-servizi.

V. IMPLEMENTAZIONE

Nel seguente paragrafo si vogliono approfondire le implementazioni delle funzionalità più interessanti.

A. SENTIMENT ANALYSIS

L'analisi del sentimento consente di predire fluttuazioni nel mercato, migliorare il servizio offerto e analizzare l'andamento delle visite in un luogo. È stato realizzato categorizzando le recensioni inserite degli utenti in *positive*, *negative*, *miste* o *neutrali*.

Abbiamo ricavato le recensioni per i lidi appartenenti ad una determinata località; tali testi sono stati successivamente analizzati e raggruppati da un algoritmo di *machine learning*. Infine, è stato ricavato il grafico dell'andamento del sentimento. In particolare, è stato utilizzato il metodo *detect_sentiment* () offerto dal servizio AWS Comprehend.

B. ALGORITMO DI SUGGESTION

L'obiettivo dell'algoritmo è suggerire all'utente le proposte in base ai suoi desideri e alle sue abitudini, prestando anche attenzione alla reputazione dei vari lidi.

Fase di selezione:

L'utente inserisce i dati in una form in cui esprime le sue preferenze riguardo al luogo, al budget massimo ecc … L'algoritmo calcola i prezzi offerti dai lidi considerando il periodo selezionato, i prezzi per ombrelloni, sdraio ed altre utilità. Successivamente scarta:

- i preventivi che superano una *soglia* predeterminata;
- i lidi fuori dal *raggio* di X km dalla meta selezionata.

Inizialmente la *soglia* e il *raggio* sono fissi. Se l'utente ha prenotato più di tre volte utilizzando la nostra piattaforma, ha accumulato dati a sufficienza nella sua storia-utente che possono essere usati per creare delle statistiche personalizzate. Queste informazioni accumulate ci dicono:

1) Quanto dista mediamente il prezzo finale dal budget indicato. Questo dato fa variare la *soglia* del prezzo ed è molto utile quando l'utente tende a sovrastimare il budget.

2) Quanto dista la meta finale da quella indicata nella form. Se l'utente tende a rimanere molto vicino alla città selezionata ha senso ridurre il *raggio* di ricerca.

3) Caratteristiche medie ricercate nei lidi. Ogni stabilimento balneare ha con sé diverse informazioni: presenza di bar, ristorazione, palestra ecc… Memorizzando le caratteristiche dei lidi visitati, possiamo capire quanto sia compatibile un lido con l'utente.

I punti (1) e (2) servono per discriminare se scartare o no una proposta. Il punto (3) serve per la fase successiva.

Fase di ordinamento:

Le richieste selezionate vengono ulteriormente processate per stabilire in che ordine andranno mostrate all'utente. Con le caratteristiche medie ricercate nei lidi (punto 3), vengono create 3 liste in base al livello di correlazione: *alto*, *medio* o *basso*. Ogni lista viene, a sua volta, ordinata con il voto del lido (ricavato dalle recensioni degli altri utenti) e mostrata all'utente.

Potevamo sfruttare un algoritmo di machine learning già implementato come abbiamo fatto per l'analisi del sentimento. Abbiamo deciso di procedere con un algoritmo ad-hoc per i nostri bisogni. Tuttavia, la funzionalità di *suggestion* è estremamente lenta a causa:

- della complessità dell'algoritmo – rimangono aperte eventuali possibilità di ottimizzazione che, per questione di tempo, non sono state approfondite.
- dell'interazione con gli altri micro-servizi per il calcolo dei preventivi e dei voti dei lidi. Questa comunicazione è sincrona (con gRPC) e presenta molta più latenza rispetto ad una locale.

C. TRANSAZIONI

Tutte le scritture e le eliminazioni multiple nelle tabelle su dynamoDB sono racchiuse in transazioni: eseguono secondo un approccio "*ALL* or *NOTHING*".
A tale scopo, viene utilizzato un "low-level service client" offerto da boto3 sul quale invochiamo il metodo *transact_write_items* (). La segnatura del metodo prevede in input un array di *TransactWriteItem*: in ognuno di questi elementi è contenuto l'oggetto di più basso livello da eseguire (Put, Delete o Update). La *transact_write_items* () è un'operazione di scrittura sincrona che può raggruppare fino a 100 *action requests*. È importante notare che le operazioni possono coinvolgere più tabelle ma non account AWS differenti.
Le transazioni sono state utilizzate ampiamente durante la stesura codice, ma si sono rivelate essenziali nel contesto particolare del pattern SAGA. È emblematica, a tal proposito, la delete dell'account (che ricordiamo essere inserita in una

SAGA). L'operazione coinvolge tutte le tabelle in dynamoDB in modo da non tralasciare nessun dato relativo all'utente in questione. Se queste operazioni di eliminazione non fossero state inserite in un approccio transazionale, l'intero pattern SAGA avrebbe potuto portare l'applicazione in uno stato incoerente e, quindi, non corretto.
Infine, qualsiasi operazione che coinvolge scritture multiple, per aver successo è eseguita secondo le proprietà logiche ACID tipiche delle transazioni.

VI. LIMITAZIONI

A. *Deploy su docker compose*

L'applicazione è pensata per il deploy su una sola macchina quindi, come framework per l'orchestrazione dei container, è stato utilizzato Docker Compose. Abbiamo riscontrato problemi nella replicazione del micro-servizio stateful che, al suo interno, ha un database privato per memorizzare lo stato. Tale micro-servizio viene istanziato con l'ausilio dei volumi (per creare i database nei container) : alla sua replicazione, i diversi container condividono il volume, dunque, non abbiamo implementato un meccanismo di consistenza dei database. Usando la replicazione attraverso l'opzione

--scale microservice_name = num_instances (1)

non siamo riusciti a evitare la condivisione del volume.

Non abbiamo potuto partizionare lo stato tra le repliche in quanto non trovavamo una suddivisione dei dati che riuscisse a soddisfare le richieste. Per questo abbiamo pensato di replicare lo stato.

B. *Flask*

Flask non permette di gestire diverse sessioni contemporaneamente quindi non è possibile avere due utenti su uno stesso browser. Per testare la concorrenza abbiamo sfruttato il deploy sulle macchine virtuali per poi connetterci contemporaneamente dai nostri pc.

VII. ORGANIZZAZIONE DEL CODICE

Il codice è organizzato in cartelle contenenti i file dei vari micro-servizi. All'esterno c'è il file di docker-compose che si occupa, tra le altre cose, di creare dei volumi in modo da comunicare ai container il codice sorgente del micro-servizio da eseguire.
Nella cartella dello specifico servizio c'è:

- Un main che raccoglie tutti i metodi contattabili con gRPC;
- Un file per l'interazione con il database (che ha anche la logica relativa alle transazioni);
- Uno o più file per la comunicazione asincrona che contengono la logica di inizializzazione delle code di messaggi e i metodi di callback da eseguire alla ricezione;
- Eventuali file contenenti logiche di appoggio.

Infine, la definizione dei protocol buffer è stata accorpata nel file grpc.proto nella cartella "*proto*" (seguendo l'esempio di Google: GoogleCloudPlatform/microservices-demo: Sample cloud-native application with 10 microservices showcasing Kubernetes, Istio, and gRPC. (github.com)). Abbiamo inserito il file protoc.sh che compila grpc.proto e sposta i file generati nelle cartelle dei micro-servizi.

Tutti i metodi sono stati opportunamente documentati in modo da rendere più comprensibile la lettura del codice.