Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

Politecnico di Torino

Creazione e test di un sistema di raccomandazione in un servizio web basato su microservizi

Relatore:
Prof. Luca Ardito

Candidato:
Renato Di Lorenzo

Correlatore:
Prof. Maurizio Morisio

Dicembre 2021
Torino

Sommario

Il progetto DME, Digital Management of Events, nasce con lo scopo di realizzare una piattaforma digitale per l'organizzazione di eventi di vario tipo (matrimoni, compleanni, conferenze, concerti, etc.). Realizzato con la collaborazione del Politecnico di Torino, dell'Università degli Studi di Bari e dell'azienda Linear System, il progetto è finanziato dalla Regione Puglia ed ha come obiettivo iniziale l'ingresso nel mercato dell'organizzazione di eventi regionale, per poi espandersi sia nel territorio nazionale, che europeo. La piattaforma DME si propone di superare le limitazioni di prodotti esistenti, fornendo un unico strumento in grado di coinvolgere tutti gli stakeholder del processo di organizzazione degli eventi, dai fornitori agli utenti finali. Nei primi mesi di progettazione sono stati sviluppati due macro moduli: Front End e Back End. La piattaforma si basa su un'architettura a microservizi, e sull'utilizzo di altre tecnologie innovative come distributed ledger per smart contract.

Nel contesto sopra descritto, l'obiettivo del presente lavoro di tesi è lo sviluppo di un microservizio recommender system da integrare nell'architettura DME esistente. Tale microservizio ha lo scopo di fornire agli utenti suggerimenti sui fornitori di servizi a cui rivolgersi in fase di organizzazione di un evento, nell'ottica di offrire un'esperienza personalizzata. I suggerimenti sono basati sulle valutazioni espresse da utenti con gusti simili (Collaborative Filtering), nonché sui criteri di selezione espressi dall'utente. Il microservizio si interfaccia con i servizi di back end DME esistenti, al fine di reperire le valutazioni degli utenti e per le funzionalità di autenticazione e autorizzazione, attraverso API REST.

L'uso di un'architettura a microservizi consente la scelta autonoma e indipendente di linguaggi e tecnologie per ciascun modulo dell'applicazione. L'implementazione è realizzata in Java, attraverso il framework Spring Boot e l'utilizzo di un database a grafo Neo4j. L'algoritmo di Collaborative Filtering utilizzato appartiene alla famiglia denominata Slope One. Il microservizio offre una API REST per consentirne l'interrogazione da parte del front

end o di altri servizi back end.

Il risultato ottenuto rappresenta una base di partenza che può essere ulteriormente migliorata, attraverso l'uso di algoritmi più complessi e tecniche di Machine Learning. Ciò sarà reso più semplice grazie alla flessibilità offerta dall'utilizzo di un'architettura a microservizi, che rende più agile lo sviluppo indipendente di moduli separati.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Contesto

DME (Digital Management of Events) è un progetto di ricerca e sviluppo finanziato dalla Regione Puglia e sviluppato attraverso la collaborazione di molteplici entità, tra cui il Politecnico di Torino, l'Università degli Studi di Bari e l'azienda Linear System. L'obiettivo è la realizzazione di una piattaforma digitale innovativa per l'organizzazione di eventi.

Il mercato dell'organizzazione di eventi, specialmente in ambito professionale, è in crescita a livello mondiale poiché gli eventi rappresentano un'opportunità di marketing e promozione aziendale per molte compagnie. Tipicamente, l'organizzazione di un evento coinvolge molteplici aziende e fornitori di servizi, oltre a richiedere la gestione di diversi aspetti, che vanno dalla pianificazione alla comunicazione.

In questo contesto si va a posizionare il progetto DME, che prevede la realizzazione di una piattaforma digitale che implementi un modello di business che coinvolge sia gli utenti finali che i fornitori di servizi. Infatti, l'obiettivo è quello di fornire strumenti di pianificazione e organizzazione, ma anche un marketplace per far incontrare domanda e offerta. Tale piattaforma dovrà offrire diverse funzionalità innovative, quali il supporto agli Smart Contract attraverso un Distributed Ledger e la capacità di fornire suggerimenti agli utenti in base alla propria storia di utilizzo della piattaforma, attraverso un Recommender System. Lo sviluppo, inoltre, prevede l'uso di tecnologie e metodologie moderne, quali architetture a microservizi, metodologia Agile e DevOps e utilizzo di infrastrutture cloud. Il presente lavoro riguarda la progettazione e lo sviluppo del microservizio Recommender che sarà parte del layer di back end dell'applicazione DME.

Motivazioni

Le architetture a microservizi sono un argomento di particolare interesse, affrontato in parte nel corso di Applicazioni Internet, durante il percorso di laurea. Pertanto, la proposta di contribuire allo sviluppo di un'applicazione basata su questa tecnologia rappresenta un'opportunità di affrontare in modo più approfondito questo tema.

Obiettivi

I principali obiettivi sono i seguenti:

- Acquisire conoscenze sull'impiego di architetture a microservizi per lo sviluppo di applicazioni web, comprenderne i vantaggi, gli svantaggi, le problematiche comuni e le possibili soluzioni.
- Acquisire conoscenze sui Recommender System, in particolare sul ruolo che svolgono nei servizi web, sulle diverse tipologie esistenti e su possibili implementazioni.
- Sviluppare un microservizio che impieghi tecniche di raccomandazione da integrare nell'architettura a microservizi dell'applicazione DME.
- Acquisire dimestichezza con metodologie di sviluppo Agile attraverso la collaborazione tra l'azienda Linear System e gli studenti e ricercatori del Politecnico di Torino, in un contesto di lavoro da remoto.

Tecnologie

Uno dei benefici delle architetture a microservizi è la possibilità di scegliere, per ciascun microservizio, le tecnologie da adottare, in autonomia e indipendenza dal resto dell'applicazione. Per il microservizio Recommender, lo strato di persistenza è stato realizzato con Neo4j, un DBMS a grafo. Il livello applicativo è stato realizzato in linguaggio Java, attraverso il framework Spring. L'algoritmo di raccomandazione implementato è la versione Weighted della famiglia di algoritmi di Collaborative Filtering nota come Slope One.

Contenuti

La prima parte dell'elaborato affronta, dal punto di vista teorico, i principali concetti che riguardano i Recommender System e le architetture a microservizi. Per quanto riguarda i primi, nel capitolo 2 sono illustrati gli utilizzi

e le principali tipologie. Vengono trattati principalmente gli algoritmi di collaborative filtering, tra cui la famiglia Slope One, di cui si è realizzata un'implementazione. Il capitolo 3 affronta le architetture a microservizi, confrontandole con l'approccio monolitico, esponendo i vantaggi e trattando le varie problematiche che esse presentano, indicando possibili soluzioni. La seconda parte dell'elaborato tratta l'implementazione del microservizio di Recommender. Vengono illustrate le tecnologie scelte, tra cui il DBMS a grafo Neo4j, il framework Java Spring e i relativi moduli. Viene poi descritta l'implementazione e i test effettuati su di essa.

Capitolo 2

Recommender System

2.1 Introduzione

I recommender system o recommendation system sono strumenti software progettati con l'obiettivo di suggerire oggetti agli utenti.

Con il termine oggetto (*item*) si fa riferimento a ciò che il sistema deve consigliare all'utente. Ad esempio, i suggerimenti possono riguardare film da vedere, musica da ascoltare, prodotti da acquistare, notizie da leggere, persone da aggiungere agli amici in social network, ecc. Spesso i recommender system sono progettati in modo da essere specifici per il tipo di oggetti che devono suggerire, in modo da massimizzare l'efficacia dei suggerimenti.

L'esigenza di Recommender System in grado di orientare gli utenti nella scelta di prodotti da acquistare è nata con il proliferare dell'e-commerce. Cataloghi sempre più vasti e alternative sempre più numerose comportano una maggiore difficoltà nel processo di decisione, quindi il rischio di effettuare una scelta sbagliata, sopraffatti dalla vastità dell'offerta. Infatti, se avere più opzioni tra cui scegliere è indubbiamente un fatto positivo, averne troppe può risultare controproducente.

In generale, lo scopo principale di un recommender system è quello di orientare gli utenti privi di esperienza o conoscenze necessarie nella scelta tra molti oggetti alternativi. I suggerimenti sono di solito personalizzati, quindi mirati ai gusti individuali di ciascun utente. Infatti, fornire suggerimenti impersonali su oggetti più popolari non è un problema particolarmente complesso da affrontare, ma non è detto che tutti gli utenti apprezzino gli stessi oggetti. È proprio nella capacità di fornire suggerimenti personalizzati, rispondenti ai gusti del singolo utente, che risiede la complessità ed è su questo aspetto che si concentra la ricerca sui Recommender System.

Al giorno d'oggi, per gli utenti internet è molto comune imbattersi in Recommender System. Tante categorie di servizi online di uso quotidiano ne beneficiano: e-commerce, servizi di streaming di contenuti multimediali, siti di notizie, social network, ecc.

Tipicamente, i suggerimenti sono rappresentati sotto forma di graduatoria e ogni oggetto ha un punteggio. Gli oggetti che hanno punteggio più elevato sono quelli ritenuti più adatti alle esigenze e alle preferenze dell'utente, da parte del Recommender System. Affinché il sistema sia in grado di fornire suggerimenti personalizzati, però, c'è bisogno della capacità di creare un modello o profilo di ciascun utente, contenente le proprie preferenze.

Le preferenze degli utenti sono ricavate a partire da informazioni raccolte dal Recommender System nelle precedenti interazioni tra l'utente e il sistema. Le tecniche con cui queste si ricavano, però, variano in base al sistema. Possono essere considerate preferenze espresse esplicitamente dall'utente, come l'assegnazione di un voto, oppure possono essere ottenute interpretando azioni compiute dall'utente. Ad esempio, l'apertura della pagina di un articolo da parte dell'utente può essere vista come una manifestazione implicita di interesse verso quell'articolo.

La prima forma di Recommender System sviluppata è quella basata sull'approccio collaborativo conosciuto come *Collaborative Filtering*. L'idea di base è quella di sfruttare le informazioni raccolte sulle preferenze dell'intera comunità di utenti: suggerire oggetti piaciuti ad utenti con gusti simili. Se in passato altri utenti hanno apprezzato gli stessi oggetti dell'utente corrente, è probabile che questo possa essere interessato ad altri oggetti apprezzati da questi utenti.

2.1.1 Motivazioni e Ruolo

Le motivazioni che spingono le aziende ad adottare un Recommender in un servizio o applicazione internet possono essere:

- Incremento delle vendite. Se gli utenti ricevono suggerimenti validi sono più propensi ad acquistare gli oggetti che visualizzano, quindi aumenta quello che viene definito *tasso di conversione*.

- Diversificazione degli oggetti venduti. Un Recommender System può consigliare agli utenti oggetti di loro gradimento, ma al contempo poco popolari, quindi meno facilmente scopribili autonomamente.

- Incremento del tasso di soddisfazione. Gli utenti sono più propensi ad usare il servizio se questo è in grado di suggerire oggetti di loro gradimento.
- Incremento del tasso di fedeltà. Più gli utenti interagiscono con il servizio, più sono le informazioni che il Recommender System è in grado di raccogliere sui loro gusti. Di conseguenza, i suggerimenti futuri saranno sempre migliori.
- Raccolta di informazioni sulle preferenze degli utenti al fine di offrire un servizio migliore.

Gli utenti, invece, sono motivati ad accettare i suggerimenti dei Recommender System se li ritengono validi. Quindi vanno conciliate le esigenze dei fornitori con quelle degli utenti.

I Recommender System possono avere diversi ruoli all'interno di sistemi informativi. Ad esempio, in alcuni casi basta avere consigli su alcuni oggetti di possibile gradimento, mentre in altri si vogliono trovare tutti gli oggetti che rispondono alle esigenze dell'utente. Oppure il Recommender System si può limita ad evidenziare alcuni oggetti in una lista più ampia. Pertanto, sono state sviluppate molteplici tecniche che si basano su svariati tipi di informazioni.

2.1.2 Fonti di informazioni e dati

I Recommender System hanno bisogno di raccogliere dati per poter generare i suggerimenti. Questi possono riguare sia gli oggetti da suggerire che gli utenti a cui effettuare i suggerimenti. A seconda del tipo di Recommender System, si può decidere di raccogliere e utilizzare alcune informazioni piuttosto che altre, tra le tante possibili. Le tecniche più semplici si limitano a raccogliere informazioni di base, come i voti espressi dagli utenti, mentre altre richiedono una conoscenza più profonda sia degli utenti che degli oggetti.

Alla base del processo di raccolta di informazioni, si possono identificare tre entità principali:

- *Oggetti*. Sono le entità che vengono consigliate agli utenti. Ciascun oggetto ha un valore per l'utente e il compito del Recommender System è quello di stimare questa utilità.
  Ciascun oggetto ha delle proprietà che il Recommender System può considerare per generare i suggerimenti. Ad esempio, un libro ha caratteristiche come l'autore, il genere, l'anno di pubblicazione, ecc.

- *Utenti*. Sono coloro i quali ricevono i suggerimenti. Per fornire suggerimenti personalizzati, il sistema deve conoscere informazioni sugli utenti, quindi deve essere in grado di generare un modello di ciascun utente. Nel caso più semplice, queste sono le valutazioni espresse in passato dall'utente sugli oggetti, ma i sistemi possono anche considerare altre caratteristiche del'utente, come l'età, il genere, ecc.

- *Transazioni*. Sono le interazioni tra utenti e sistema che vengono collezionate dal sistema per poter ricavare informazioni utili per la generazione di suggerimenti. Il caso più comune di transazione è una valutazione espressa da un utente su un oggetto.
  Una valutazione può essere numerica, binaria (positiva o negativa), ordinale, oppure unaria. Una valutazione unaria può essere l'acquisto o anche la semplice visualizzazione di un oggetto. Per esempio, se un utente apre la pagina della descrizione di un prodotto a partire dai risultati di una ricerca, anche se non effettua l'acquisto, si può interpretare l'azione come una valutazione implicita, quindi come una manifestazione di interesse.
  Le valutazioni implicite non richiedono alcuno sforzo aggiuntivo da parte dell'utente, ma non è garantito che siano del tutto accurate. L'acquisto di un prodotto non garantisce il gradimento dello stesso da parte dell'acquirente.

2.1.3 Tipologie

I Recommender System possono essere classificabili in diverse categorie, in base al modo con cui vengono generati i suggerimenti.

Collaborative Filtering

L'approccio collaborativo, o Collaborative Filtering, è il primo e più semplice tra quelli utilizzati per realizzare Recommender System. Gli oggetti che vengono suggeriti sono quelli apprezzati da utenti con gusti simili. L'idea di base è che, se un utente in passato ha avuto gusti simili ad altri utenti, questi probabilmente continueranno ad avere gusti simili in futuro. La somiglianza tra I gusti degli utenti è determinata dalle valutazioni che essi assegnano agli oggetti. Non si fa, invece, uso di informazioni sulle caratteristiche degli oggetti. I Recommender System più diffusi sono basati su questo approccio.

Il vantaggio principale di questa tipologia di algoritmi è che non richiedono informazioni sugli oggetti da consigliare. Infatti, non sempre è semplice

raccogliere e mantenere questo tipo di informazioni per tutti gli oggetti del catalogo. Tuttavia, l'approccio collaborativo richiede che vi sia un certo numero di valutazioni espresse dall'utente corrente su oggetti passati e da altri utenti su altri oggetti. In assenza di valutazioni, ad esempio su nuovi oggetti del catalogo, è impossibile fornire suggerimenti.

Approccio Content-Based

L'approccio basato sui contenuti, o Content-Based, consiste nel suggerire oggetti simili a quelli che l'utente ha già apprezzato. La somiglianza tra gli oggetti è determinata sulla base di caratteristiche comuni. Ad esempio, per i film si possono valutare attributi come genere, regista, attori, ecc. Una valutazione espressa su un oggetto può essere considerata come una valutazione sugli attributi dell'oggetto, quindi si può generare un profilo dell'utente sulla base di queste valutazioni e generare suggerimenti di oggetti, assegnando loro un punteggio di affinità con le caratteristiche del profilo dell'utente.

Questo approccio non richiede valutazioni da parte di molti utenti per poter generare suggerimenti, quindi è valido anche nel caso di oggetti nuovi, a differenza dell'approccio collaborativo. Le complessità, però, risiedono nel trovare il modo di raccogliere le informazioni sugli oggetti e costruire i profili degli utenti. Se si fa affidamento a sistemi manuali per l'inserimento delle caratteristiche di oggetti, si rischia di incorrere in errori, oltre ad essere un processo lungo e costoso. Quindi si tende ad usare sistemi automatizzati, ad esempio di analisi semantica. Invece, per la generazione dei profili degli utenti, si usano tecniche di machine learning, che imparano sulla base delle interazioni precedenti.

Approccio demografico

L'approccio demografico prevede suggerimenti sulla base di informazioni demografiche dell'utente, ad esempio l'età, la lingua, il paese, ecc.

Approccio Knowledge-Based

L'approccio basato sulla conoscenza, o Knowledge-Based, invece di impiegare informazioni ricavate dalle precedenti interazioni tra utente sistema, sfrutta la conoscenza del dominio per determinare gli oggetti con caratteristiche corrispondenti alle esigenze dell'utente. Quindi si determina una somiglianza tra i requisiti espressi dall'utente e le caratteristiche degli oggetti.

Un esempio sono i Recommender basati sui vincoli, o *Constraint-Based*, che richiedono all'utente, in modo interattivo, informazioni sui requisiti e le usano per generare suggerimenti. Per rendere più personalizzati i suggerimenti, può anche essere richiesto di esprimere un punteggio che indichi l'importanza per l'utente di ciascun requisito espresso.

Approcci Ibridi

Gli approcci ibridi puntano a combinare i benefici e superare le limitazioni dei singoli meccanismi sopra elencati, al fine di generare suggerimenti più precisi. Ciò comporta una maggiore complessità dovuta all'integrazione di tecniche e algoritmi separati.

Esistono tre modalità per integrare informazioni contestuali nei Recommender System:

- Architettura monolitica: viene realizzato un unico Recommender System che impieghi diverse tecniche per trattare i dati in ingresso.
- Architettura in parallelo: diversi Recommender System lavorano in modo indipendente e i risultati vengono successivamente combinati in qualche modo.
- Architettura in serie: i risultati di un Recommender System diventano input per il successivo Recommender System.

2.2 Collaborative Filtering

I Recommender System basati sull'approccio collaborativo (*Collaborative Filtering*) sono la tipologia più diffusa e la prima ad essere stata sviluppata. Per questa ragione, sono stati nel corso del tempo studiati e perfezionati estensivamente. Esistono molteplici algoritmi basati su questo approccio.

Il principio alla base è quello di utilizzare le informazioni raccolte in precedenza su opinioni o comportamenti dell'intera comunità di utenti, al fine di determinare quali oggetti suggerire all'utente corrente.

In generale, un algoritmo di collaborative filtering prende in ingresso una matrice contenente le valutazioni che ogni utente ha assegnato ad ogni oggetto e restituisce, per ciascun oggetto, un valore numerico che rappresenta una previsione del gradimento che l'utente può avere per esso. Solitamente si seleziona un certo numero di oggetti col valore di gradimento predetto più elevato, in modo da formare una lista di oggetti suggeriti.

L'approccio collaborativo può essere suddiviso in due classi: *neighborhood-based* e *model-based*. La prima categoria prevede l'uso diretto delle valutazioni archiviate dal recommender system, al fine di generare i suggerimenti. In particolare, si usano per trovare utenti o oggetti simili a quello in questione e questi prendono il nome di neighbor.
Il collaborative filtering model-based, invece, prevede prima la generazione di un modello a partire dalle valutazioni nel sistema, utilizzando tecniche di data mining / machine learning. I suggerimenti saranno forniti applicando il modello.

2.2.1 User-based Nearest Neighbor

Il primo approccio alle raccomandazioni collaborative è quello conosciuto come user-based nearest neighbor.

Partendo dall'assunzione che utenti che hanno avuto gusti simili in passato continuino ad averli in futuro e che i gusti degli utenti siano stabili nel tempo, il primo passaggio consiste nell'identificare, dato un utente, gli altri utenti che hanno espresso gusti simili in passato, quindi gli utenti adiacenti più vicini (nearest-neighbors). In particolare, dato un utente u, i nearest-neighbors di u sono gli utenti v, diversi da u, che hanno la maggiore somiglianza con u.

Per stabilire il numero di vicini più adiacenti da prendere in considerazione si può procedere in due modi: stabilire una soglia di somiglianza o stabilire un numero k di vicini più adiacenti, i *k-nearest-neighbors* (*k-NN*). Se la soglia è troppo bassa si selezionano troppi vicini, se è troppo alta se ne selezionano troppo pochi e di conseguenza si riduce il numero di oggetti su cui si può fare una previsione. In modo simile, la scelta di k influenza la qualità dei risultati. Ciò va tenuto in considerazione quando si effettua la scelta di questi valori.

Successivamente, per ciascun oggetto non ancora valutato dall'utente corrente, viene calcolata una stima sulla base delle valutazioni degli utenti vicini. Una possibile misura di somiglianza tra gli utenti è il *coefficiente di correlazione di Pearson*. È stato dimostrato empiricamente che questa misura sia tra le più performanti se usata per determinare la somiglianza tra utenti.
Dati:

- l'insieme di utenti $U = \{u_1, \ldots, u_n\}$,
- l'insieme di oggetti $P = \{p_1, \ldots, p_n\}$,
- la matrice R di valutazioni $r_{i,j}$ con i utente e j oggetto
- $\overline{r_a}$ e $\overline{r_b}$ valutazioni medie degli utenti a e b

Il coefficiente di correlazione di Pearson definisce la misura di somiglianza tra l'utente a e l'utente b come:

$$sim(a,b) = \frac{\sum_{p \in P} (r_{a,p} - \overline{r_a})(r_{b,p} - \overline{r_b})}{\sqrt{\sum_{p \in P} (r_{a,p} - \overline{r_a})^2} \sqrt{\sum_{p \in P} (r_{b,p} - \overline{r_b})^2}}$$

I valori ottenuti variano da $+1$ (forte correlazione positiva) a -1 (forte correlazione negativa). Il coefficiente di Pearson tiene conto del fatto che alcuni utenti tendono ad assegnare voti più alti, mentre altri tendono a darli più bassi, rendendo il confronto più equo.

Una volta determinati gli utenti più vicini, si può effettuare la stima della valutazione per gli oggetti non ancora valutati dall'utente corrente. Ciò va fatto tenendo però conto della somiglianza, in modo che utenti con somiglianza maggiore abbiano maggiore peso di quelli meno simili, ad esempio, nel modo seguente:

$$pred(a,p) = \overline{r_a} + \frac{\sum_{b \in N} sim(a,b) * (r_{b,p} - \overline{r_b})}{\sum_{b \in N} sim(a,b)}$$

dove a è l'utente, p è l'oggetto e $\overline{r_a}$ è la valutazione media di a.
A questo punto, gli oggetti con valutazione stimata più alta possono essere suggeriti all'utente.

Il coefficiente di correlazione di Pearson non tiene conto che spesso ci sono oggetti che tendono a piacere a molti utenti, quindi dovrebbero essere meno rilevanti nello stabilire la somiglianza tra due utenti. Una possibile soluzione è trasformare le valutazioni in modo da ridurre l'importanza degli elementi universalmente apprezzati.

Un altro problema del coefficiente di Pearson è che non si tiene conto della quantità di elementi in comune tra due utenti e, se questi sono pochi, la somiglianza può essere casuale e le previsioni possono risultare inaccurate.

2.2.2 Item-based nearest neighbor

Quando il numero di utenti e oggetti è molto elevato, quindi la matrice diventa estremamente grande, le tecniche user-based iniziano a diventare inadeguate a generare suggerimenti in tempo reale. Per questa ragione, sono state sviluppate le tecniche item-based che consentono di pre-effettuare una parte del processo computazionale, semplificando la generazione di suggerimenti in tempo reale.

Invece di calcolare la somiglianza tra gli utenti, si calcola la somiglianza tra gli oggetti, sempre sulla base delle valutazioni che hanno ricevuto dagli utenti.

Una misura valida per determinare la somiglianza tra oggetti è la somiglianza del coseno (*cosine similarity*). Si tratta di una misura per determinare la somiglianza tra vettori, in questo caso i vettori dei voti assegnati ad ogni oggetto.
Dati $\vec{a}$ e $\vec{b}$ vettori di voti assegnati agli oggetti a e b, la misura di somiglianza del coseno è definita come:

$$sim(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|}$$

Questa misura può essere migliorata in modo da tenere conto della differenza tra i voti medi di ciascun utente. Tale variante prende il nome di *adjusted cosine similarity*. Dato U, l'insieme degli utenti che hanno valutato sia a che b, questa misura prevede di sottrarre dalle valutazioni degli utenti le loro valutazioni medie:

$$sim(a, b) = \frac{\sum_{u \in U} (r_{u,a} - \overline{r_u})(r_{u,b} - \overline{r_u})}{\sqrt{\sum_{u \in U} (r_{u,a} - \overline{r_u})^2} \sqrt{\sum_{u \in U} (r_{u,b} - \overline{r_u})^2}}$$

Calcolata la somiglianza tra gli oggetti, si può procedere a calcolare le stime delle valutazioni per ciascun oggetto, per l'utente corrente.

$$pred(u, p) = \frac{\sum_{i \in rateditems(u)} sim(i, p) * r_{u,i}}{\sum_{i \in rateditems(a)} sim(i, p)}$$

Per velocizzare il calcolo dei suggerimenti in tempo reale al fine di migliorare i tempi di risposta, si può effettuare a priori una parte della computazione, generando *matrici di somiglianza* tra gli oggetti: per ogni coppia di oggetti contengono il valore di somiglianza precalcolato. In fase di generazione di suggerimenti si fa riferimento ai valori della matrice per calcolare le somme pesate delle valutazioni degli oggetti vicini. Dovendo considerare solo oggetti valutati dall'utente, il tempo richiesto per queste computazioni è ragionevole per poterle fare in tempo reale.

La precomputazione delle matrici di somiglianza si adatta meglio all'approccio item-based, poiché le somiglianze tra oggetti sono molto più stabili all'aggiungersi di valutazioni, rispetto a quelle tra utenti.

2.2.3 Slope One

Slope One è una famiglia di algoritmi di item-based Collaborative Filtering, pubblicata nel 2005 in un articolo di Daniel Lemire e Anna Maclachlan intitolato "Slope One Predictors for Online Rating-Based Collaborative Filtering".

La motivazione dietro lo sviluppo di questi algoritmi è l'esigenza di avere una metodologia per implementare il Collaborative Filtering con le seguenti caratteristiche:

- facilità di implementazione e testing;
- possibilità di aggiornare al volo le predizioni aggiungendo nuovi voti;
- velocità di interrogazione;
- possibilità di fornire suggerimenti anche ad utenti con poche valutazioni espresse;
- ragionevole accuratezza di risultati, ma senza compromettere la scalabilità.

Il principio su cui si basano è definito dagli autori come *differenziale di popolarità* tra gli oggetti per gli utenti: si determina la differenza di gradimento che c'è tra due coppie di oggetti per un utente e la si usa per fare una predizione per un altro utente.

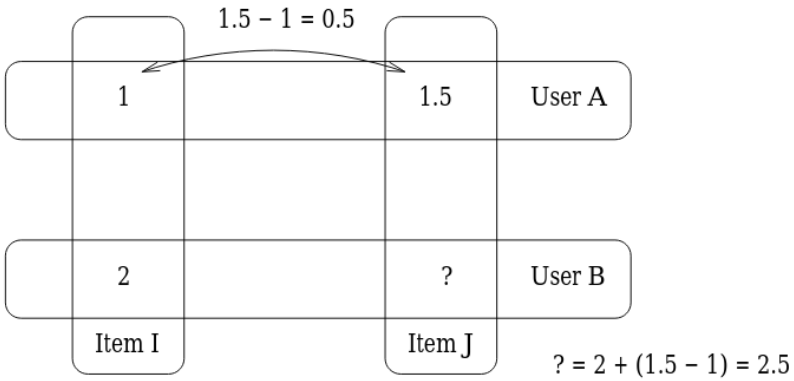

Figura 2.1. Differenziale di popolarità

Nell'esempio in figura 2.1, per User A la differenza di valutazione tra gli oggetti J e I è uguale a 0,5. La predizione della valutazione dell'oggetto J per l'utente B, data la valutazione che esso ha dato all'oggetto I pari a 2, può essere ottenuta calcolando $2 + 0{,}5 = 2{,}5$.

Quindi, per un oggetto non ancora valutato da un utente, si considerano le differenze tra i voti dell'oggetto e quelli di altri oggetti dati da altri utenti,

e si fa una media. Queste differenze medie vengono usate per effettuare la predizione del voto per l'oggetto e l'utente in questione.

Le predizioni avvengono secondo una formula del tipo $f(x) = x+b$, che è il motivo per cui prende il nome di Slope One. Questa regressione è più semplice della regressione lineare poiché ha un solo parametro libero (b), mentre x è la variabile che rappresenta le valutazioni. Ciononostante, è dimostrato che i risultati ottenuti sono competitivi con altri meccanismi memory-based.

Dati:

- χ: insieme di valutazioni,
- $u \in \chi$: vettore di valutazioni di un utente,
- i e j: oggetti,
- $S_{j,i}(\chi)$: sottoinsieme di χ contenente i vettori di valutazioni degli utenti che hanno valutato sia i che j,
- $card\,(S_{j,i}(\chi))$: numero di utenti che hanno valutato sia i che j,
- u_i e u_j: elementi di u, le valutazioni degli oggetti i e j

si definisce la deviazione media dell'oggetto i rispetto all'oggetto j come:

$$dev_{j,i} = \sum_{u \in S_{j,i}(\chi)} \frac{u_j - u_i}{card\,(S_{j,i}(\chi))}$$

Si considerano solo gli utenti che hanno valutato sia i che j.

La deviazione media per ogni coppia di oggetti, una volta calcolata, può essere archiviata in una matrice aggiornabile rapidamente con le nuove valutazioni.

La predizione del voto dell'oggetto j per l'utente u può essere calcolata nel seguente modo:

$$P^{S1}(u)_j = \bar{u} + \frac{1}{card\,(R_j)} \sum_{i \in R_j} \mathrm{dev}_{j,i}$$

dove R_j è l'insieme di oggetti con valutazione in u, diversi da j. In questa versione ottimizzata, non si tiene in considerazione il voto dell'utente u ad ogni oggetto i, ma solo il valore medio $\bar{u}$.

Slope One non considera, per ogni coppia di oggetti, la quantità di utenti che hanno valutato entrambi. Più questi sono, maggiore sarà l'accuratezza

della predizione. Per questo motivo, la variante pesata di Slope One tiene in considerazione la cardinalità nel calcolo della predizione:

$$P^{wS1}(u)j = \frac{\sum_{i \in S(u)-\{j\}} (\mathrm{dev}_{j,i} + u_i)\, card\,(S_{j,i}(\chi))}{\sum_{i \in S(u)-\{j\}} card\,(S_{j,i}(\chi))}$$

Un'altra variante di Slope One chiamata Bi-Polar, sia nel calcolo della predizione che nel calcolo delle deviazioni, prevede la separazione tra oggetti valutati positivamente e oggetti valutati negativamente.

Le valutazioni effettuate dagli autori su dataset di esempio dimostrano che l'accuratezza dei suggerimenti forniti da Slope One sono in linea con altri schemi memory-based, nonostante i vantaggi prestazionali.

Capitolo 3

Architetture a microservizi

3.1 Introduzione

Nell'ambito della progettazione e dello sviluppo di servizi web, le architetture a microservizi rappresentano sicuramente uno degli sviluppi di maggiore successo degli ultimi anni.

Il principio di base è quello di scorporare quelle che tradizionalmente sarebbero le diverse funzionalità di una singola applicazione monolitica in tante piccole applicazioni indipendenti, ma in grado di comunicare fra loro attraverso API. Ci si riferisce a queste applicazioni, appunto, con il nome di microservizi. Ad esempio, in un'applicazione di e-commerce, i vari aspetti da gestire ciascuno attraverso un proprio microservizio potrebbero essere il catalogo, gli ordini, i pagamenti, le notifiche e-mail, ecc.

Le architetture a microservizi comportano una serie di vantaggi rispetto al tradizionale approccio monolitico, in particolare in termini di rapidità di sviluppo, scalabilità, resilienza, facilità di deployment e versatilità nella scelta delle tecnologie. Tuttavia, l'impiego di un'architettura a microservizi comporta anche degli svantaggi e criticità che riguardano sia l'organizzazione dello sviluppo che aspetti tecnici dovuti ad una maggiore complessità.

3.2 Approccio Monolitico

Nelle tradizionali applicazioni monolitiche è pratica comune quella di usare un approccio modulare, suddividendo in diversi moduli i diversi aspetti della

logica di business. Quindi si definisce una suddivisione dell'applicazione dal punto di vista logico, ma i vari moduli che compongono l'applicazione vengono compilati formando un monolite, il cui deployment avviene in un'unica operazione. Ad esempio, un'applicazione Java EE viene impacchettata in un unico file WAR.

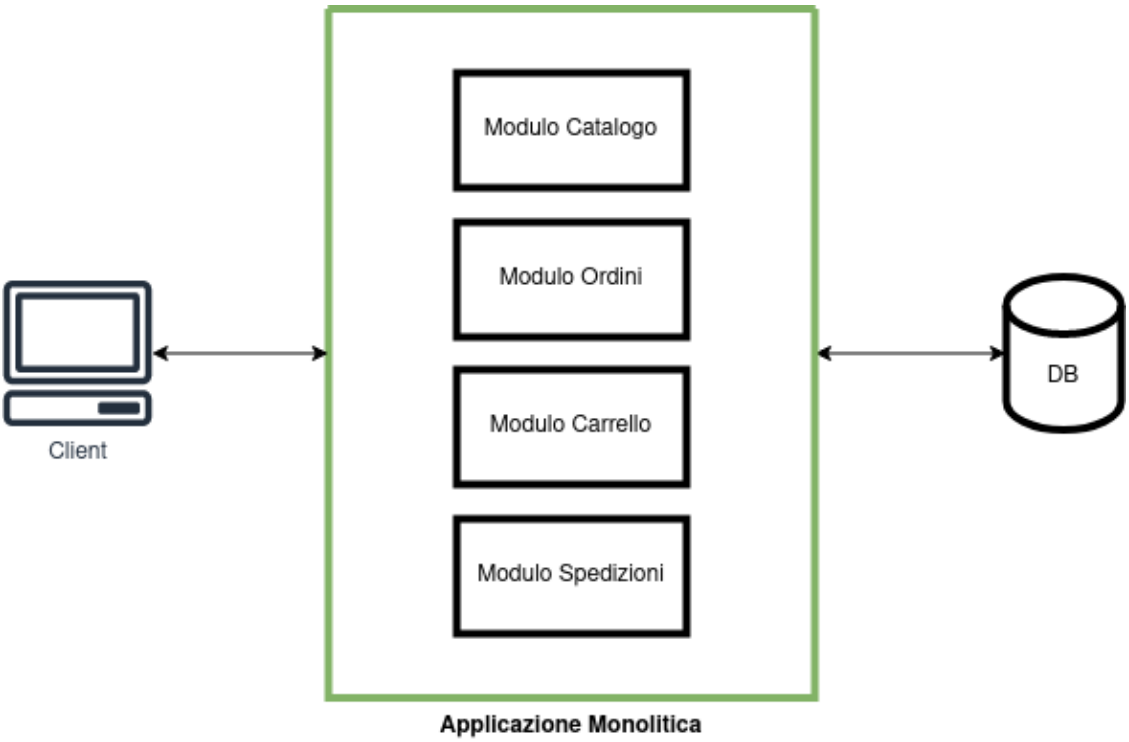

Figura 3.1. Esempio di architettura monolitica

Ciò rende questo tipo di approccio vantaggioso dal punto di vista della semplicità di testing end-to-end e deploying. La scalabilità si può ottenere attraverso l'utilizzo di più istanze dell'applicazione gestite da un load balancer.

Con il passare del tempo, però, se l'applicazione ha successo e continua ad essere sviluppata, la complessità tende ad aumentare significativamente e, con essa, diversi svantaggi diventano progressivamente più accentuati.

Quando l'applicazione diventa eccessivamente grande, il codice risulta di difficile comprensione per gli sviluppatori che subentrano. Anche per gli sviluppatori esperti, però, può essere problematico apportare modifiche al codice senza causare danni. Inoltre, con l'aumentare della complessità c'è anche il rischio che la modularità iniziale si perda.

Un'applicazione molto grande tendenzialmente richiede molto tempo per la compilazione e il deployment, e questo ha impatto sulla produttività degli sviluppatori. Anche il solo tempo di avvio può diventare importante al punto da rallentare significativamente lo sviluppo. In generale, la maggiore complessità rappresenta un ostacolo alla distribuzione continua e allo sviluppo agile.

Dal punto di vista organizzativo, al crescere della complessità diventa più complicato suddividere il lavoro in team che si occupino di aspetti diversi dell'applicazione. Infatti, questi non possono lavorare in modo autonomo e indipendente, in quanto i deployment hanno effetto sull'intera applicazione ed è necessaria coordinazione.

Modifiche che impattano una piccola parte del codice richiedono comunque la compilazione e il deployment dell'intero monolite. Questo può comportare lunghe interruzioni del servizio, specialmente se i tempi di avvio sono molto lunghi, ed eventuali problemi riguardanti singoli moduli avrebbero ugualmente effetto sull'intera applicazione.

La scalabilità di un'applicazione monolitica può avvenire in una sola dimensione. Impiegando più copie dell'applicazione dietro un load balancer si possono gestire più transazioni, ma tutte le copie devono avere accesso a tutti i dati, quindi sono difficili da scalare all'aumentare della mole di dati. Inoltre l'intera applicazione deve essere replicata, non è possibile scalare solo i moduli che richiedono maggiore computazione.

Infine, l'approccio monolitico è molto limitante dal punto di vista della scelta di tecnologie e linguaggi. Una volta scelto uno stack all'inizio del processo di sviluppo, questo è destinato a rimanere costante nel tempo. Se si vogliono adottare nuove tecnologie, a meno che queste non prevedano l'interoperabilità con quelle in uso, è necessaria la riscrittura dell'intera applicazione e questa migrazione non può avvenire in modo graduale, un modulo per volta.

Le complicazioni sopra elencate, al crescere dell'applicazione, comportano quello che viene definito *Monolithic Hell*: l'applicazione diventa troppo grande e ne risentono la facilità di sviluppo, l'affidabilità, la scalabilità, ecc.

3.3 Approccio a microservizi

Il modello di architettura a microservizi nasce per superare le limitazioni imposte dall'approccio monolitico. Il concetto di modularità già presente già nelle applicazioni monolitiche, viene ulteriormente accentuato: servizi

individuali, di dimensioni ridotte ed interconnessi, sostituiscono i moduli del servizio monolitico.

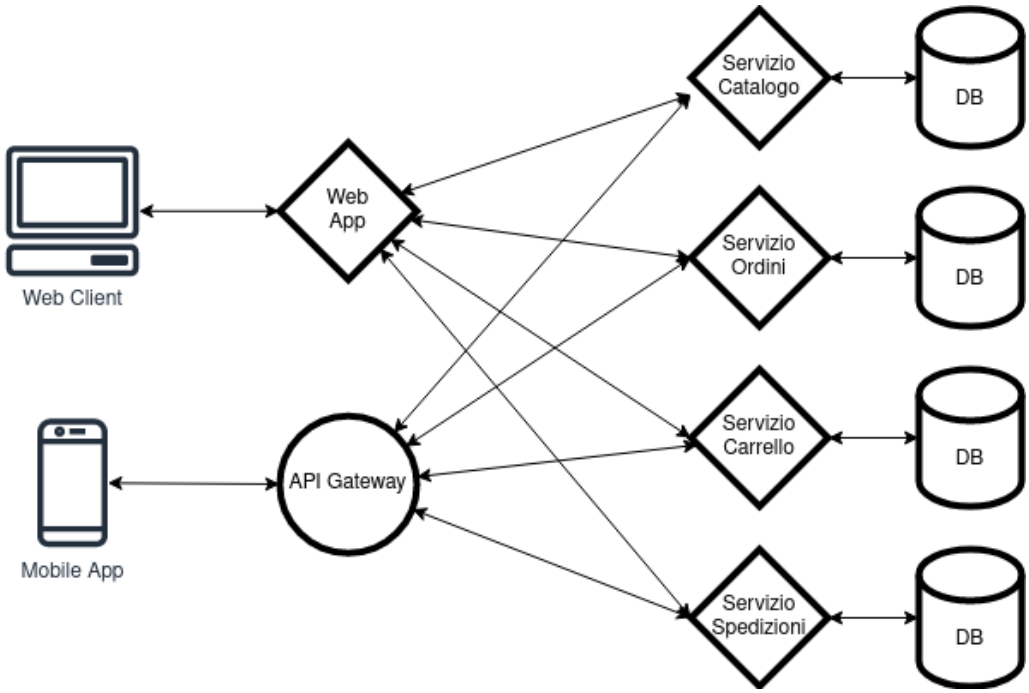

Figura 3.2. Esempio di architettura a microservizi

Tipicamente, un microservizio gestisce un'area funzionale, cioè un insieme di funzionalità logiche tra loro attinenti. Queste funzionalità sono esposte agli altri micorservizi attraverso una API REST o altri meccanismi di comunicazione. Quindi, un micorservizio può consumare le API offerte da altri micorservizi e offrire a sua volta una propria API. Si dice che i microservizi di un'applicazione sono tra loro debolmente accoppiati (*loosely coupled*): un microservizio può dipendere da una API offerta da un altro microservizio, ma non dalla sua implementazione. Quindi, l'implementazione di un microservizio può essere completamente alterata senza conseguenze sugli altri, fintanto che l'API esposta non varia.

In aggiunta alle funzionalità di back end, anche un'applicazione web che svolge la funzionalità di interfaccia utente diventa un microservizio a sé stante, in grado di scambiare dati con i microservizi di back end attraverso le API da loro esposte. Anche per l'interfaccia utente può aver senso, in alcuni casi, la suddivisione delle funzionalità in più microservizi. Se ci sono diverse classi di utenti, che hanno accesso a funzionalità del tutto diverse dell'applicazione,

può risultare opportuno che ognuno abbia un servizio di front end ad essa dedicato.

In un'ipotetica applicazione di e-commerce, una possibile architettura a microservizi è quella illustrata nel in figura 3.2. Nel diagramma è rappresentato anche un *API Gateway* che costituisce un unico punto di ingresso per le API offerte dai vari microservizi di back end. In questo esempio, esso è impiegato per gestire i client mobili, ma in generale un API Gateway può essere adoperato per molteplici scopi.

Una importante caratteristica delle architetture a microservizi è la relazione tra l'applicazione e i dati. Invece di avere un unico database centralizzato, infatti, ogni microservizio è dotato di un proprio database autonomo, nel quale sono archiviati i dati ad esso relativi. Questo comporta una inevitabile duplicazione di dati che sono di interesse di più microservizi, ma è necessario per garantire l'accoppiamento lasco che caratterizza i microservizi. Se, infatti, più microservizi dovessero accedere allo stesso database, una modifica allo schema da parte di uno avrebbe impatto su tutti gli altri. Un ulteriore beneficio derivante dall'avere database esclusivi del singolo microservizio è che ogni microservizio può essere progettato per usare il DBMS più adatto ai propri requisiti e alle funzionalità che deve offrire. Si ha, perciò, quella che viene definita *persistenza poliglotta*.

3.4 Scalabilità e architetture a microservizi

Con il termine scalabilità si definisce la capacità di un servizio ad adattarsi all'aumentare della mole di lavoro aggiungendo risorse. Secondo il modello del *Cubo della Scalabilità*, rappresentato in figura 3.3, un'applicazione può essere scalata lungo tre assi.

L'asse X è quello della *duplicazione orizzontale*. Questo tipo di scalabilità si può realizzare anche con applicazioni monolitiche e consiste nel replicare l'applicazione in più istanze dietro un *load balancer*. Questo ha il compito di smistare il traffico tra le varie copie dell'applicazione in modo da distribuire il carico. Questo tipo di scalabilità è semplice e veloce da realizzare, ma, come accennato nei paragrafi precedenti, tutte le copie dell'applicazione devono poter accedere a tutti i dati. Ciò comporta un costo notevole dovuto alla necessità di replicare i dati. Inoltre, non si ha scalabilità dal punto di vista organizzativo, l'applicazione resta monolitica.

L'asse Y è quello della *decomposizione funzionale*. Consiste nel suddividere le funzionalità dell'applicazione per verbi, cioè casi d'uso, oppure per

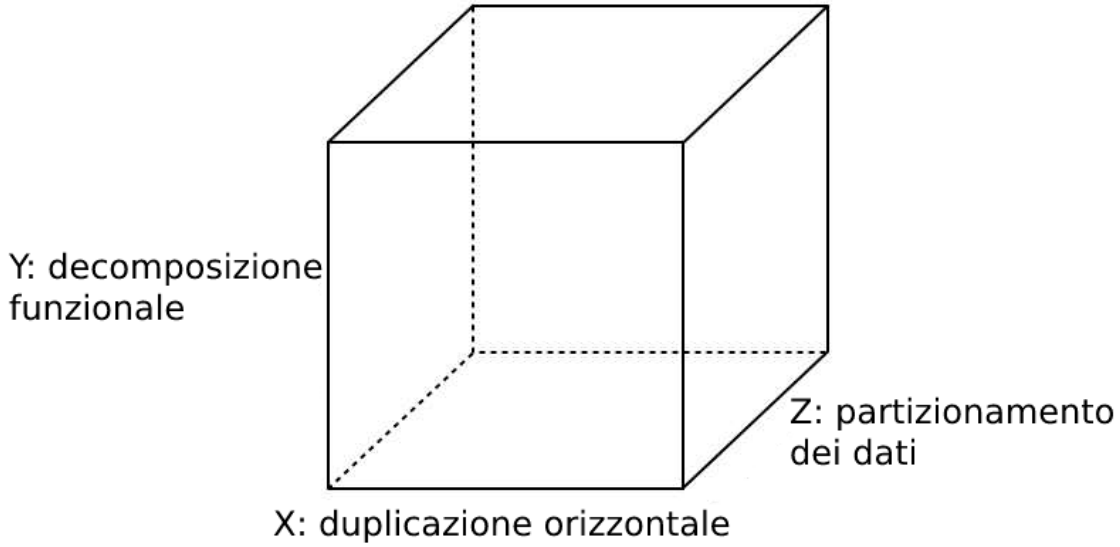

Figura 3.3. Modello del Cubo della scalabilità

nomi, cioè entità. L'utilizzo di architetture a microservizi è essenzialmente un'applicazione di questo tipo di scalabilità.

L'asse Z, infine, rappresenta il *partizionamento dei dati*. È simile alla scalabilità lungo l'asse X, con più repliche della stessa applicazione. La differenza è che il carico di lavoro è diviso in base ai dati, poiché ogni copia gestisce uno specifico sottoinsieme dei dati. Il load balancer deve quindi smistare le richieste in base ai relativi dati. Questo tipo di scalabilità è generalmente usato per le basi di dati.

Spesso le applicazioni usano una combinazione dei tre approcci per ottenere scalabilità, in quanto i singoli microservizi possono essere duplicati o anche partizionati.

3.5 Vantaggi delle architetture a microservizi

Oltre ai benefici in termini di scalabilità discussi nel paragrafo precedente, le architetture a micorservizi offrono diversi vantaggi. La maggior parte delle limitazioni e degli svantaggi dell'approccio monolitico dipendono dalla complessità che le applicazioni possono raggiungere al crescere delle funzionalità. Impiegando un'architettura a microserivizi, il problema della complessità viene superato suddividendo l'applicazione in pezzi più piccoli e quindi più semplici da gestire individualmente. In questo modo si garantisce la modularità

e si risolvono molti dei problemi delle grandi applicazioni monolitiche.

La dimensione ridotta di ciascun microservizio rende il codice più semplice da comprendere e, di conseguenza, da modificare. Anche il testing di ciascun microservizio risulta più semplice, ma non il testing end-to-end.

I tempi di compilazione e di avvio sono ridotti e questo facilita la distribuzione continua, alla base dello sviluppo agile.

Aumenta anche la capacità di isolamento dei guasti. Infatti, un eventuale bug introdotto in un microservizio ha effetto solo sulle funzionalità che questo offre, invece di rendere potenzialmente inutilizzabile l'intera applicazione.

Suddividendo l'applicazione in microservizi, diventa possibile suddividere anche il lavoro degli sviluppatori in team indipendenti. Fintanto che i contratti tra i servizi sono fissati, ciascun team può lavorare autonomamente e indipendentemente dagli altri.

Infine, si ha un'elevata flessibilità nello scegliere le tecnologie e i linguaggi da utilizzare. Ogni team può progettare ed implementare il proprio microservizio con le tecnologie che ritiene più adatte, senza essere legato alle scelte altrui. Anche se un team decide di riprogettare il proprio microservizio usando nuove tecnologie, ciò può avvenire indipendentemente dal resto dell'applicazione.

3.6 Problematiche delle architetture a microservizi

Come qualsiasi altra tecnologia, le architetture a microservizi non sono una soluzione perfetta, bensì presentano degli svantaggi e gli sviluppatori che scelgono di adottarle devono affrontare una serie di problematiche.

La principale fonte di problemi nelle architetture a microservizi è la natura distribuita. Quando i moduli di un'applicazione diventano ciascuno un servizio a sé stante, bisogna affrontare il problema della comunicazione tra diversi processi, stabilendo degli appositi meccanismi. Inoltre, l'applicazione deve essere in grado di gestire l'eventualità di fallimenti parziali, ad esempio quando non si riesce a comunicare con un particolare microservizio. In contrapposizione all'approccio monolitico, dove si ha a che fare con semplici invocazioni tra diversi moduli della stessa applicazione, questo rappresenta sicuramente uno scenario più complesso da gestire.

Oltre che tecnico, il problema è anche organizzativo, in quanto è necessaria coordinazione tra i vari team. Specialmente nel caso in cui si devono aggiungere funzionalità ad un'applicazione esistente, e queste coinvolgono

più microservizi tra loro dipendenti, è necessario pianificare e coordinare correttamente queste operazioni in modo da accertarsi che il deployment dell'aggiornamento di un microservizio avvenga solo dopo l'aggiornamento dei microservizi da cui esso dipende.

L'implementazione di operazioni che coinvolgono più microservizi è più complessa, specialmente considerando che ognuno di essi mantiene il proprio database e si vuole mantenere una certa consistenza tra i dati distribuiti.

Sebbene il testing individuale di ciascun microservizio sia facilitato dalle dimensioni ridotte, il testing end-to-end dell'intera applicazione, quindi delle interazioni tra i vari microservizi, è sicuramente più complesso.

Infine, le architetture a microservizi sono meno efficienti dal punto di vista del consumo di memoria. Visto che ogni servizio è un'applicazione isolata, eseguita in un un proprio container o in una propria virtual machine, molte delle dipendenze risultano replicate.

Nessuna di queste problematiche rappresenta un ostacolo insormontabile ed esistono soluzioni comprovate e pronte da essere adottate. Tuttavia, è opportuno tenere presente che gli innegabili benefici di questo tipo di architettura vengono ad un costo che la rende conveniente solo per applicazioni complesse e in costante evoluzione.

3.7 Interazioni tra microservizi

Suddividendo l'applicazione in microservizi, questi dovranno necessariamente interagire tra loro per scambiarsi i dati. Nel caso di applicazioni monolitiche, le interazioni che avvengono sono invocazioni di funzioni o metodi tra diversi moduli dello stesso processo, mentre in questo caso devono avvenire tra processi diversi (*IPC*).

Le interazioni tra processi possono essere uno-a-uno, cioè processate da un solo microservizio, oppure uno-a-molti, cioè processate da più microservizi. Inoltre possono essere sincrone, con il client bloccato in attesa di risposta, o asincrone, con il client non bloccato. In questo contesto, con il termine "client" si intende anche essere un microservizio che interagisce con altri microservizi, inviando richieste/notifiche. Meccanismi di interazione uno-a-uno:

- *Request/Response* (sincrono): il client effettua una richiesta e attende una risposta entro un certo tempo.
- *Notification* (asincrono): il client effettua una richiesta senza aspettarsi risposta.

- *Request/Async response*: il client effettua una richiesta e la risposta del servizio avviene in modo asincrono. Il client non si blocca in attesa.

Meccanismi uno-a-molti:

- *Publish/Subscribe* (asincrono): il client pubblica una notifica e zero o più servizi subscriber la ricevono
- *Publish/Async* response: il client pubblica una notifica e zero o più servizi subscriber la ricevono e rispondono entro un certo tempo.

Questi meccanismi possono essere usati singolarmente o in combinazione, a seconda delle necessità.

I dati scambiati possono essere in formato testuale o binario. Il vantaggio di usare un formato testuale come *JSON* o *XML* è la leggibilità umana, oltre al fatto che ogni valore è descritto da un nome o una chiave. D'altra parte, il formato testuale è meno prestante poiché va effettuato il parsing e più verboso.

L'API esposta da un microservizio è un contratto tra esso e i client (inclusi altri microservizi). Il formato di API dipende dal meccanismo di IPC che si sceglie di usare. Quando si vuole modificare l'API di un servizio esistente, va tenuto conto che non tutti i client saranno aggiornati allo stesso momento. È buona norma progettare client e servizi in modo che siano robusti a piccoli cambiamenti delle API. Quindi, client e servizi devono essere in grado di lavorare con versioni API leggermente diverse, ad esempio usando valori di default per attributi mancanti nelle richieste e ignorando attributi aggiuntivi (non attesi) nelle risposte. Quando, invece, i cambiamenti apportati all'API di un servizio sono sostanziali, è buona norma supportare entrambe le versioni, finché tutti i client non sono stati aggiornati. Nel caso di API HTTP REST, il client può specificare la versione dell'API nell'URL della richiesta e il servizio risponde di conseguenza.

La natura distribuita delle architetture a microservizi comporta la necessità di considerare e gestire i fallimenti parziali. Per qualsiasi motivo, ad esempio per sovraccarico o per problemi di rete, un servizio può risultare non disponibile o non in grado di rispondere alle richieste in tempi ragionevoli. In queste circostanze, i client devono evitare di restare bloccati indefinitamente, altrimenti ben presto i thread disponibili si esauriscono. Ciò è particolarmente valido quando si usano meccanismi sincroni. Ci sono varie soluzioni per gestire i fallimenti parziali. Innanzitutto si devono usare timeout e imporre limiti alle richieste pendenti in uscita. Poi esiste il pattern denominato *Circuit Breaker*: tenendo traccia del tasso di errori nelle richieste in uscita, si

determina se è il caso di effettuarne delle altre o aspettare un certo intervallo di tempo. Inoltre, è opportuno prevedere valori di default da usare come fallback in assenza di risposta.

3.7.1 IPC asincrona

La comunicazione tra processi asincrona avviene attraverso lo scambio di messaggi. Quando un servizio riceve un messaggio può decidere di rispondere con un altro messaggio o meno, ma, in entrambi i casi, il client che ha inviato il messaggio originale non resta bloccato in attesa di una risposta. Se si attende una risposta, il client non fa assunzioni sui tempi.

I messaggi sono composti da intestazione e corpo e vengono scambiati attraverso canali. I canali possono essere punto-punto (comunicazione uno-a-uno) o publish/subscribe (comunicazione uno-a-molti). Nel pattern publish/subscibe, i client non inviano i messaggi a destinatari specifici, ma li inviano in canali ai quali altri servizi possono iscriversi. Ad esempio, quando su un sito di e-commerce un utente completa un ordine, il servizio degli ordini può inviare un messaggio nel canale "Ordini completati" e altri servizi iscritti, come quello della logistica, lo ricevono.

Esistono molteplici meccanismi di messaggistica asincrona con caratteristiche diverse, che fanno uso di protocolli proprietari o standard (come *AMQP*).

La comunicazione attraverso messaggi presenta diversi vantaggi. Consente il disaccoppiamento tra client e servizi, infatti i client possono inviare messaggi in canali, non a servizi specifici. I messaggi possono essere messi in un buffer ed essere gestiti in un secondo momento, non essendo richiesta risposta immediata. È supportata la comunicazione uno-a-molti in aggiunta a quella uno-a-uno.

D'altro canto, la comunicazione asincrona comporta una maggiore complessità operazionale dovuta alla presenza del sistema di messaggistica, che va configurato e mantenuto, al pari degli altri componenti dell'applicazione. Inoltre c'è una maggiore complessità di implementazione, specialmente nel caso in cui sono previste risposte, in quanto queste, essendo asincrone, vanno correlate alle richieste attraverso un identificativo di correlazione.

3.7.2 IPC sincrona

Con il meccanismo di *Request/Response* sincrono, il client resta bloccato in attesa della risposta, poiché c'è l'aspettativa che questa arrivi entro un certo

tempo.

Il protocollo più comune per questo tipo di comunicazione è REST. *REST (Representational State Transfer)*, è uno stile per la creazione di API ed usa quasi sempre HTTP (e varianti sicure) come protocollo. Con REST vengono definite risorse che rappresentano solitamente entità della logica di business dell'applicazione. L'accesso alle risorse e la loro manipolazione vengono richiesti attraverso i verbi HTTP come GET, PUT e POST, che specificano l'azione. A diverse risorse corrispondono diversi endpoint dell'API, cioè diversi URL. Ad esempio, con "POST /users" si richiede la creazione di un utente con le informazioni specificate nel corpo della richiesta, e con "GET /users/<id>" si richiedono le informazioni su uno specifico utente.

Quando una API segue i principi REST viene definita *RESTful*, ma, secondo il Modello di Maturità ci sono quattro livelli di aderenza allo stile REST:

- Livello 0: tutte le operazioni offerte dall'API si effettuano attraverso un unico endpoint e con il verbo POST.
- Livello 1: endpoint corrispondenti alle risorse, ma solo verbo POST.
- Livello 2: verbi HTTP corrispondenti alle azioni.
- Livello 3: si segue il principio HATEOAS (Hypertext As The Engine Of Application State) secondo cui le risorse restituite devono contenere collegamenti alle azioni possibili. In questo modo, i client possono sapere con certezza quali azioni sono disponibili e i relativi URL.

L'uso di API REST basate su HTTP comporta vantaggi di semplicità, dovuti principalmente all'utilizzo di un protocollo semplice e già affermato come HTTP. Tuttavia HTTP ha limitazioni, come il solo supporto al meccanismo Request/Response, cioè ogni richiesta prevede una risposta. Inoltre non c'è buffering, quindi client e servizio devono essere contemporaneamente disponibili. Infine c'è il problema della necessità da parte del client di conoscere gli URL dei vari servizi che richiede un meccanismo di service discovery. Una alternativa a REST, meno diffusa, è *Apache Thrift*.

3.8 Service discovery

Per consumare una API offerta da un servizio, un client deve conoscere indirizzo e porta di un'istanza a cui indirizzare la richiesta. In un ambiente cloud

moderno, è probabile che ci siano più istanze dello stesso servizio, per motivi di scalabilità e ridondanza. Il numero di istanze di ogni servizio non è fisso e le loro posizioni di rete non si possono conoscere a priori. Per questo motivo, si ha la necessità di meccanismi di *Service Discovery*. Questi si dividono in due categorie: client-side e server-side.

Il meccanismo *client-side prevede* che i client siano incaricati di trovare le posizioni delle istanze di un servizio e sceglierne una secondo meccanismi di load balancing. Ciò avviene interrogando l'API offerta da un servizio che funge da registro delle istanze dei servizi. Quando un'istanza di un servizio viene creata, questa viene aggiunta al registro, sempre secondo i meccanismi previsti dall'API del servizio di registro. Tipicamente questo si occupa anche di monitorare lo stato delle istanze ed eventualmente rimuovere dal registro quelle inattive.

Questo approccio è semplice da realizzare, ma i client devono impiegare la logica per interrogare il registro ed effettuare il load balancing delle richieste. *Netflix Eureka* è un esempio di servizio di registro.

L'approccio *server-side*, oltre al registro, prevede un servizio load balancer a cui i client inviano le richieste. È questo servizio che ha il compito di consultare il servizio di registro per scoprire le istanze disponibili e successivamente sceglierne una.

Il vantaggio principale è che i client non devono implementare la logica di discovery e load balancing. Tuttavia è necessario aggiungere un servizio all'architettura, il load balancer. Questo può essere offerto dal fornitore di servizi cloud.

Il servizio di registro deve essere sempre raggiungibile, per questo spesso si usa un cluster di server e un protocollo di replicazione per mantenere consistenti le informazioni tra di essi. Un modo per ottenere gli indirizzi del registro è quello di usare il DNS e scegliere l'istanza più vicina.

La registrazione e rimozione delle istanze al registro può avvenire da parte delle istanze stesse, quando vengono create e distrutte. Ciò è concettualmente semplice, ma è un ulteriore compito che si affida ai vari servizi. L'alternativa è introdurre un ulteriore servizio esterno *Registrar*, responsabile di tenere traccia delle istanze tramite polling o sottoscrizione ad eventi e comunicarne lo stato al servizio di registro. Si rimuove complessità dai singoli servizi, ma si aggiunge un servizio all'architettura. Questo può essere offerto dal fornitore di servizi cloud.

3.9 API Gateway

Un *API Gateway* è un componente spesso presente nelle architetture a microservizi. Si tratta di un servizio che offre ai client un unico punto di ingresso per il back end dell'applicazione, evitando la comunicazione diretta tra i client e i vari microservizi.

Quando le applicazioni sono complesse, una semplice operazione di un client può comportare il coinvolgimento di decine, se non centinaia, di servizi. Una mole elevata di richieste da parte del client è difficile da gestire, sia dal punto di vista della complessità di sviluppo che dell'efficienza in termini di utilizzo della rete. Se poi i microservizi usano protocolli di comunicazione e formati di messaggi poco adatti ad essere implementati in applicazioni web o mobili, si rende necessario un meccanismo di traduzione. Queste ragioni rendono l'utilizzo di un API gateway necessario.

Un API Gateway può svolgere essenzialmente tre compiti:

- *Inoltro*: le richieste che passano per il gateway vengono inoltrate alle API dei rispettivi microservizi di back end.
- *Composizione*: se una richiesta del client si traduce in più richieste verso diversi microservizi, il gateway compone le risposte ricevute da essi in un'unica risposta da fornire al client.
- *Traduzione* tra protocolli: se per la comunicazione tra microservizi si usano protocolli non compatibili con il client, il gateway funge da traduttore.

Un vantaggio che si ricava dall'utilizzo è quello di nascondere la struttura interna dell'applicazione, quindi i servizi che la compongono. Si crea, perciò, un disaccoppiamento tra l'architettura interna dell'applicazione e quella che è esposta ai client. In questo modo si può alterare, se ritenuto opportuno, l'architettura dell'applicazione senza avere impatto sui client. Inoltre si riduce la complessità dei client, che si limitano a parlare con un solo servizio, e, grazie alla minore granularità dell'API esposta, si riduce il numero di richieste che i client effettuano.

Se ritenuto opportuno, si possono avere anche diversi gateway per diverse tipologie di client, oppure lo stesso gateway può esporre API diverse a client diversi. Un uso comune dei gateway è quello di offrire una API univoca per client e servizi di terze parti, invece di consentire l'accesso diretto ai servizi di back end. Infatti, sui client di terze parti non si ha alcun controllo e risulta necessario disaccoppiare l'API a cui hanno accesso da quella interna.

Il gateway può implementare i meccanismi di sicurezza, load balancing e caching.

L'introduzione di un API Gateway nell'architettura dell'applicazione richiede ovviamente lo sviluppo e il mantenimento di un ulteriore componente. Ricevendo tutte le richieste, c'è il rischio che diventi un collo di bottiglia per l'applicazione, quindi deve essere progettato in modo da essere leggero e scalabile. A questo proposito, diventa quasi necessario l'impiego di tecniche di programmazione asincrona non bloccante.

Per facilitare la composizione di API, che comporta l'esecuzione di molteplici chiamate, sia in serie che in parallelo, è consigliabile usare la programmazione reattiva. I meccanismi tradizionali di programmazione asincrona, basati sull'uso delle callback, sono poco comodi da usare quando si devono combinare molteplici chiamate asincrone, causando quello che è conosciuto come *Callback Hell*.

3.10 Gestione dei dati

Le applicazioni monolitiche generalmente fanno uso di un singolo DBMS, spesso relazionale, per la persistenza di tutti i dati. Questi sono facilmente interrogabili e supportano transazioni ACID:

- *Atomicità*: una transazione è atomica, cioè indivisibile.
- *Consistenza*: sono rispettati tutti i vincoli di integrità.
- *Isolamento*: le transazioni contemporanee sono indipendenti tra loro, come se avvenissero in serie.
- *Durabilità*: i risultati delle transazioni sono persistenti.

In un'architettura a microservizi, invece, ogni servizio gestisce i propri dati privati, attraverso i DBMS ritenuti più adatti, e li espone attraverso le API. Questa caratteristica, pur comportando i vantaggi illustrati in precedenza, introduce il problema della consistenza tra i vari microservizi. Infatti, secondo il *Teorema* , non è possibile avere simultaneamente Disponibilità e Consistenza tra i vari nodi. Ad esempio, un servizio che si occupa di gestire gli ordini, prima di autorizzare un ordine, deve accertarsi con il servizio che gestisce l'inventario che l'oggetto sia disponibile. Ciò non può avvenire in una singola transazione ACID, essendo database diversi gestiti da servizi diversi.

In aggiunta a questo, c'è il problema che spesso si vogliono mostrare dati provenienti da più servizi, quindi c'è bisogno di un meccanismo di join a livello di applicazione e non sempre ciò è banale.

L'utilizzo di un'architettura ad eventi, dove i servizi inviano messaggi al verificarsi di operazioni sui propri dati e gli altri servizi interessati li ricevono, può essere una soluzione per implementare transazioni tra più servizi. La transazione viene scomposta in una serie transazioni locali, riguardanti un singolo servizio, il cui esito viene comunicato attraverso un messaggio in un apposito canale. Le risorse oggetto della transazione vengono messe in uno stadio intermedio, in attesa che le altre transazioni si completino.

Riprendendo l'esempio precedente, il servizio degli ordini crea l'ordine in uno stato di attesa e lo notifica. Il servizio inventario verifica gli oggetti disponibili, riserva gli oggetti nella quantità specificata dall'ordine e invia la notifica. A questo punto il servizio degli ordini può cambiare lo stato dell'ordine in confermato e inviare la notifica. Infine, il servizio dell'inventario conferma la quantità aggiornata di oggetti.

Se, ad un certo punto, una transazione locale dovesse fallire, emettendo un evento di tipo diverso, i microservizi che hanno già fatto la propria transazione locale devono prevedere un meccanismo per ripristinare lo stato precedente.

Questo tipo di transazione composta da più transazioni locali, anche conosciuto con il nome di *Saga*, non è ACID, ma *BASE*, cioè segue un modello di *consistenza eventuale*, per cui si ha consistenza al termine di tutte le transazioni locali.

L'architettura ad eventi può essere una soluzione anche per il problema dei join di dati da più microservizi, realizzando un microservizio che, ricevendo gli eventi dagli altri servizi, mantenga una vista materializzata dei dati nel formato da inviare ai client.

L'utilizzo del pattern Saga e dell'architettura ad eventi per le transazioni comporta una certa complessità di sviluppo rispetto alle transazioni ACID che si usano nelle applicazioni monolitiche. Ad esempio, devono essere previsti meccanismi per annullare gli effetti parziali delle transazioni fallite e si devono ignorare gli eventi duplicati. Inoltre, le transazioni locali e le relative emissioni di eventi devono avvenire in modo atomico, per evitare di causare inconsistenze in caso di problemi.

Un meccanismo per ottenere l'atomicità è definito *Outbox Transazionale* e consiste nell'inserire in una tabella di eventi outbox una riga relativa all'evento, specificando lo stato pendente, nella stessa transazione locale. Un processo apposito legge questa tabella, effettua la pubblicazione dell'evento

sul canale, ed esegue un'altra transazione per marcare l'evento nella tabella come pubblicato. Se l'evento risulta pubblicato, si ha la certezza che il messaggio sia stato pubblicato almeno una volta. Bisogna accertarsi che il servizio aggiunga sempre l'evento alla tabella degli eventi quando effettua una transazione locale.

Un'alternativa è il *Log Tailing*, che consiste nell'usare un log miner che osserva il log delle transazioni del database ed emette gli eventi. Quindi si separa la logica di emissione degli eventi dal resto del servizio.

Infine si può usare l'*Event Sourcing*. Con questo approccio, al posto di archiviare le entità con il proprio stato corrente, si archiviano gli eventi che hanno modificato lo stato. Quindi si può ricostruire lo stato corrente dell'entità ripetendo la sequenza di eventi. Ciò avviene usando un particolare database chiamato *Event Store*. Il problema dell'atomicità si risolve poiché non c'è più distinzione tra una scrittura nel database e la registrazione del relativo evento, ma sono la stessa cosa. Gli altri servizi possono iscriversi agli eventi pubblicati dall'Event Store. Questo meccanismo consente anche di avere una cronologia di modifiche dello stato e ricostruire lo stato ad un certo punto nel passato. Tuttavia, è necessario acquisire familiarità con un nuovo stile di programmazione e il meccanismo di interrogazione è abbastanza complesso. Per quest'ultimo problema si può realizzare un database replica in sola lettura con viste materializzate tenute aggiornate per maggiore facilità di interrogazione.

3.11 Considerazioni sull'adozione di architetture a microservizi

Le architetture a microservizi offrono una serie di benefici che le hanno rese molto popolari nel corso degli anni. Le problematiche introdotte dall'uso di microservizi, dovute principalmente alla natura distribuita, sono molteplici, ma comuni a tutte le applicazioni che implementano questa architettura. Sono quindi state sviluppate soluzioni ai principali problemi, ma la loro adozione comporta un costo in termini di complessità non indifferente. Pertanto l'uso di architetture a microservizi risulta conveniente solo per applicazioni molto grandi, dove l'approccio monolitico diventa impossibile da gestire.

Capitolo 4

Progetto DME

4.1 Scenario di riferimento

Il progetto DME (Digital Management of Events) riguarda l'organizzazione di eventi, che negli ultimi dieci anni ha avuto un forte sviluppo, dando un input importante anche nell'ambito del marketing e della comununicazione aziendale. Questo ha permesso un aumento del numero di professionisti richiesti nel settore, e ha incentivato da una parte la loro collaborazione e dall'altra un miglioramento della produttività.
Al momento attuale non è possibile stimare il valore di mercato riferito agli eventi in Italia, ma i dati nazionali mostrano un aumento negli investimenti aziendali.

Oggi, le nuove strategie di marketing, che derivano specialmente dall'ambito digitale, portano il concetto di evento ad adattarsi a nuovi contesti e interlocutori. Infatti, una concetto di recente affermazione è "live communication", che intende l'evento come parte di una più ampia strategia di comunicazione. Gli eventi diventano, parte integrante della strategia comunicativa delle aziende, permettendo una maggiore collaborazione tra brand e fruitori del servizio. Per Brand si intende una squadra molto efficiente, con condivisione di premesse strategiche e un livello di esigenza condiviso, con sfide sempre nuove e ambiziose rispetto alle precedenti, per creare un'esperienza memorabile per i propri consumatori. La progettazione degli eventi ha dei requisiti strategici, che, come altri rami della comunicazione, iniziano dal brand, dal suo ruolo e dai suoi valori. Il brand, ha il compito di capire quali sono i target del cliente, e deve cercare di realizzare un'esperienza che sia nuova e soddisfacente per esso.

Il nuovo termine "live communication", designa una maggior professionalità da parte dei fruitori del servizio, che rispetto al passato, necessitano di dotare l'evento di strumenti sempre più interconnessi tra loro, per questo motivo l'uso della tecnologia è fondamentale. Infatti, la portata dell'evento viene amplificata tramite l'uso dei social network. L'evento diventa quindi sempre più strategico, ed un elemento integrante della comunicazione aziendale, perde la sua essenza del qui e ora. Tramite i social, il tempo di vita dell'evento è molto più lungo, poiché continua a vivere nei contenuti digitali.

4.2 Tecnologie per il supporto di eventi

La stima dei dati del settore prevede un incremento dell'organizzazione di eventi, con conseguente investimento da parte dei fornitori in soluzioni più efficaci, che portano all'utilizzo di software per la gestione organizzativa. Le aziende per poter sostenere il ritmo del mercato hanno la necessità di trovare novità per i propri business. Secondo un'analisi effettuata da Frost & Sullivan, circa l'80% delle aziende che si occupano di organizzazione di eventi lo fa mediante l'utilizzo di software di gestione. Questa percentuale si prospetta in aumento nei prossimi anni, ed inoltre si prevede una maggior diffusione di soluzioni basate sull'intelligenza artificiale piuttosto che sui Big data. Il vantaggio maggiore è dato dal fatto che l'utilizzo di interfacce utente, semplici e moderne, sia agli organizzatori che ai partecipante, rende l'organizzazione più semplice. L'uso di intelligenza artificiale permette di soddisfare i clienti garantendo loro un servizio mirato, dato da un'organizzazione guidata che fornisce le indicazioni sulle azioni e le tempistiche per il raggiungimento degli obiettivi preposti. L'obiettivo della piattaforma DME è quello di fornire strumenti di pianificazione e organizzazione, ma anche un marketplace per far incontrare domanda e offerta, coinvolgendo tutti gli attori che partecipano al processo di organizzazione di eventi.

4.2.1 Benefici attesi

Sotto il punto di vista dell'azienda Linear System possiamo distinguere quattro principali vantaggi:

1. Varietà delle proprie offerte: il progetto DME permetterà a Linear System di entrare anche nel mercato dell'organizzazione di eventi, che come

abbiamo visto precedentemente, è un mercato in continua crescita. L'idea sarà quella di partire dalla regione Puglia per poi espandersi sia campo nazionale che europeo.

2. Apprendimento nuove tecnologie:
   - Distributed ledger;
   - Cloud;
   - Architetture microservizi ;
   - Produzione in modalità Agile;
   - Deployment con DevOps;
   - Machine Learning; che permetteranno all'azienda una maggiore competizione nel mercato ICT. Le tecnologie elencate verranno acquisite da Linear System, Politecnico di Torino e Università di Bari con la modalità *learning on the job*.
3. Aumento occupazionale nella filiera dell'organizzazione di eventi: La piattaforma DME richiede, per la progettazione, implementazione e test sulla piattaforma, un team dedicato, il che implica un aumento sotto il profilo occupazionale.
4. Riduzione dei consumi: Il funzionamento in cloud permette di ridurre il consumo sia per l'azienda sopra citata, sia per i suoi clienti, che possono evitare l'acquisto di attrezzature informatiche e relativi consumi energetici.

4.3 Stato dell'arte

Il progetto DME prevede l'utilizzo di tecnologie e metodologie di punta:

- *Cloud e virtualizzazione.* La piattaforma DME è progettata per operare su macchine virtuali all'interno di infrastruttura cloud. Il cloud computing consente la creazione di pool di risorse da assegnare a progetti ed è possibile fornire maggiore capacità di elaborazione assegnando ulteriori risorse, grazie ai carichi di lavoro virtualizzati.
- *Architetture a microservizi.* Le architetture a microservizi prevedono l'organizzazione dell'applicazione in un insieme di servizi elementari indipendenti. Questo comporta vantaggi in termini di scalabilità e flessibilità di sviluppo rispetto al tradizionale approccio monolitico. D'altro

canto, le architetture a microservizi presentano una serie di problematiche, tra cui la complessità di orchestrazione, la consistenza dei dati, l'overhead e la complessità dei test di integrazione. Quando si adottano architetture a microservizi è necessario adottare soluzioni tecnologiche per ovviare a questi problemi

- *Recommender System.* Grazie all'utilizzo di tecniche di machine learning la piattaforma DME analizza i dati relativi all'organizzazione di eventi passati al fine di generare proposte di fornitori e servizi agli utilizzatori futuri.

- *Produzione di servizi multicanale.* La piattaforma DME è pensata per essere fruibile da ogni tipo di dispositivo (pc, tablet, telefono) in modo da consentire la più grande facilità d'uso possibile per gli utenti finali.

- *Sviluppo Agile e DevOps.* La metodologia di sviluppo Agile prevede frequenti rilasci del software, attraverso tecniche di Integrazione Continua e Distribuzione Continua, che consistono nel testing e deployment automatico del codice. Ciò va di pari passo con l'approccio DevOps, che determina una stretta collaborazione tra sviluppatori e operativi, per rendere più veloce l'intero ciclo di vita del prodotto.

4.4 Obiettivi

Il progetto DME prevede la realizzazione di otto Obiettivi Realizzativi (OR), suddivisi tra Ricerca Industriale (3), Sviluppo Sperimentale (4) e Project Management (1):

- OR1 - Stato dell'arte delle Metodologie e delle Tecnologie.
- OR2 - Definizione delle Tecnologie e degli Standard
- OR3 - Progettazione della Piattaforma Cloud e dei servizi Software
- OR4 - Sviluppo dell'Infrastruttura cloud
- OR5 - Sviluppo della Piattaforma DME
- OR6 - Integrazione delle componenti e test di laboratorio
- OR7 - Sperimentazione sul campo con utenti finali
- OR8 - Project Management

Ciascun obiettivo è composto da una serie di attività ed ogni attività ha lo scopo di produrre un deliverable, ad esempio un documento o un prototipo.

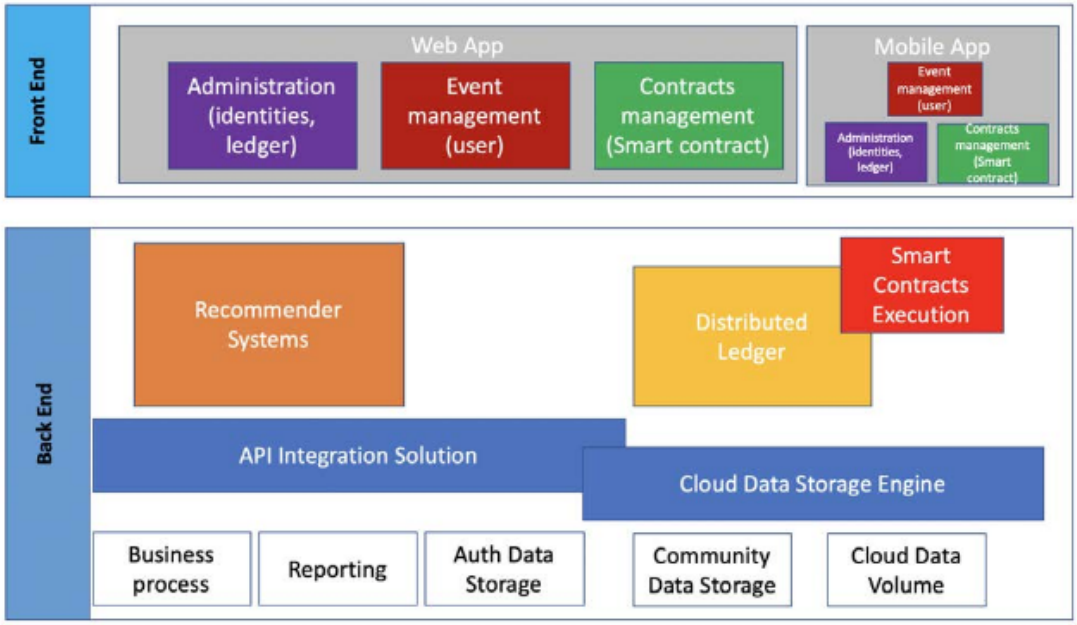

Figura 4.1. Architettura piattaforma DME

4.5 Back End

Il layer di back end dell'applicazione prevede l'impiego di un'architettura a microservizi. È previsto anche il deployment su infrastruttura cloud privata all'interno del CED (Centro Elaborazione Dati). Il modulo principale del back end implementa la maggior parte dei processi di business dell'applicazione e mantiene le informazioni relative agli utenti, ai fornitori, ecc. Inoltre, tale modulo implementa soluzioni per l'autenticazione e l'autorizzazione degli utenti. Lo sviluppo di questa parte è a carico dell'azienda Linear System.

La comunicazione tra varie componenti del layer di back end può avvenire, a seconda delle necessità e del tipo di servizi coinvolti, in modalità asincrona o sincrona. La comunicazione tra il layer di front end e quello di back end è sincrona, mediante il protocollo HTTP/HTTPS.

Questo modulo principale offre delle API interrogabili dal layer di front end e da altri servizi che compongono il layer di back end. Alcune di queste

sono private, quindi solo ad uso interno alla rete aziendale, altre saranno pubbliche, per l'uso da parte di utenti e tester.

Al modulo principale si vanno ad aggiungere altri due moduli, ciascuno realizzato attraverso un microservizio:

- Modulo Recommender System
- Modulo Distributed Ledger / Smart Contract

Il modulo Recommender System ha l'obiettivo di aiutare gli utenti nelle scelte, fornendo loro suggerimenti personalizzati. Questi possono essere basati su cronologia acquisti, valutazioni espresse, preferenze di altri utenti, ecc. Esistono vari approcci per la realizzazione di Recommender System, quali content-based, collaborative filtering, ibridi, ecc. Lo scopo è quello di avvicinare il più possibile la domanda all'offerta.

Il modulo Distributed Ledger / Smart Contract ha lo scopo di registrare in un ledger distribuito le transazioni tra clienti e fornitori, e di gestire contratti digitali dematerializzati. Un distributed ledger è un registro di transazioni immutabile, distribuito su più nodi. Ciascun nodo mantiene una copia del registro e gli aggiornamenti avvengono in modo indipendente, ma convalidati secondo un algoritmo di consenso. Una tecnologia comune per la realizzazione di distributed ledger è la Blockchain, che raggruppa le transazioni in blocchi, ciascuno con un hash che lo concatena al blocco precedente.
Per Smart Contract si intende un contratto digitale in grado di determinare automaticamente l'esecuzione delle clausole al verificarsi delle condizioni previste dal contratto.

4.6 Front End

La progettazione pone al centro dell'intero sistema l'utente, il confronto ne ha permesso uno sviluppo basato sulla User Experience Design.

La piattaforma sarà destinata a qualsiasi tipo di utente, per questo motivo il punto cardine di questo macro modulo sarà la semplicità. La UI presenta interfacce semplici e chiare che permettono di raggiungere le sezioni in modo rapido. Altro aspetto fondamentale della progettazione è dato dall'accesso multi-device, il che permette l'utilizzo della piattaforma in qualsiasi device (desktop, tablet, smartphone).

L'architettura basata su microservizi ha richiesto che la struttura comprendesse la realizzazione dei comportamenti per ciascun micro servizio. Il layer di front End permette tre tipi di registrazione: Organizzatore, Fornitore

e Invitato, e una modalità di Login che riconosce in automatico la tipologia di utente, reindirizzandolo alle sezioni ad esso dedicate, come ad esempio la sezione di creazione servizio per un fornitore, e creazione di evento per un organizzatore.

4.6.1 User experience

Gradimento degli utenti del servizio, espresso sia in modo assoluto (KPI31 voto medio dato questionario specifico alla piattaforma DME e alla sua usabilità), sia in modo relativo (KPI32 gradimento maggiore o minore della piattaforma DME rispetto ad analoghi prodotti di supporto alla gestione di eventi).

4.7 Fase di Validazione

La piattaforma DME sarà il risultato finale del progetto, su cui verranno eseguite le verifiche. L'analisi della piaattaforma potrà essere eseguita in corso d'opera sui deliverable intermedi di progetto. Le modalità di controllo saranno svolte tramite controllo documentale e tramite verifiche e misure sul prototipo. Alla conclusione dell'attività saranno resi i risultati per ciascuna attività.

4.7.1 Alpha Testing

Lo scopo degli Alpha testing è quello di rilevare Bug, anomalie e funzionamenti mancanti, in una prima fase, la seconda fase prevede un controllo della qualità ed ulteriori verifiche. Si prevede che questi vengano effettuati quando la piattaforma non è ancora completamente pronta all'uso, ma in uno stato tale che ne permetta una valutazione. I Test prevederanno diverse fasi:

- Revisione specifiche e requisiti funzionali di progettazione;
- Sviluppo piano di test
- Esecuzione del piano di test definito
- Registrazione dei difetti

Il procedimento sarà iterativo, fino a che tutti i problemi non saranno risolti.

4.7.2 Simulazione casi reali

A seguito degli Alpha Testing, verranno svolti i test con gli utenti. I test su casi reali permetteranno di valutare la soddisfazione potenziale dell'utente, attraverso una visita guidata nella piattaforma. Il fine sarà quello di capire se agli utenti piace la piattaforma DME. Gli utenti selezionati non conoscono la piattaforma, e sono esterni al progetto, questo permette una maggiore oggettività da parte della loro esperienza sulla piattaforma.

Capitolo 5

Implementazione

5.1 Analisi dei requisiti

Il processo di sviluppo è stato eseguito secondo la metodologia *Agile*. A tal proposito, nella fase iniziale non erano stati specificati requisiti formali relativi al microservizio di Recommender e, per questo motivo, si è potuto procedere in autonomia nell'interpretare il problema e produrre una soluzione. Si è scelto, quindi, di realizzare un microservizio che implementasse un algoritmo di Collaborative Filtering (Weighted Slope One) per fornire suggerimenti di fornitori agli utenti. La scelta è stata dettata dal buon rapporto tra semplicità di implementazione e prestazioni che lo caratterizza.

In un secondo momento, da parte dell'azienda che gestisce il progetto, è stato esplicitato il requisito di poter esprimere, nella richiesta di suggerimenti, anche due parametri di filtraggio dei fornitori (*codGeo* e *codCategoria*). Il servizio è stato quindi modificato per accomodare questa richiesta.

Successivamente, si è deciso che il microservizio dovesse fornire anche suggerimenti sui tipi di servizio da abbinare al tipo di evento scelto. A causa degli stringenti vincoli temporali, si è deciso di implementare questa funzionalità come prototipo, fornendo suggerimenti basati su relazioni statiche, inserite manualmente nel database. In futuro questo compito sarà eseguito sulla base delle transazioni passate, eseguite dagli utenti della piattaforma.

5.1.1 Use Case

I requisiti sopra descritti sono rappresentati nello Use Case Diagram di figura 5.1 e nelle successive tabelle.

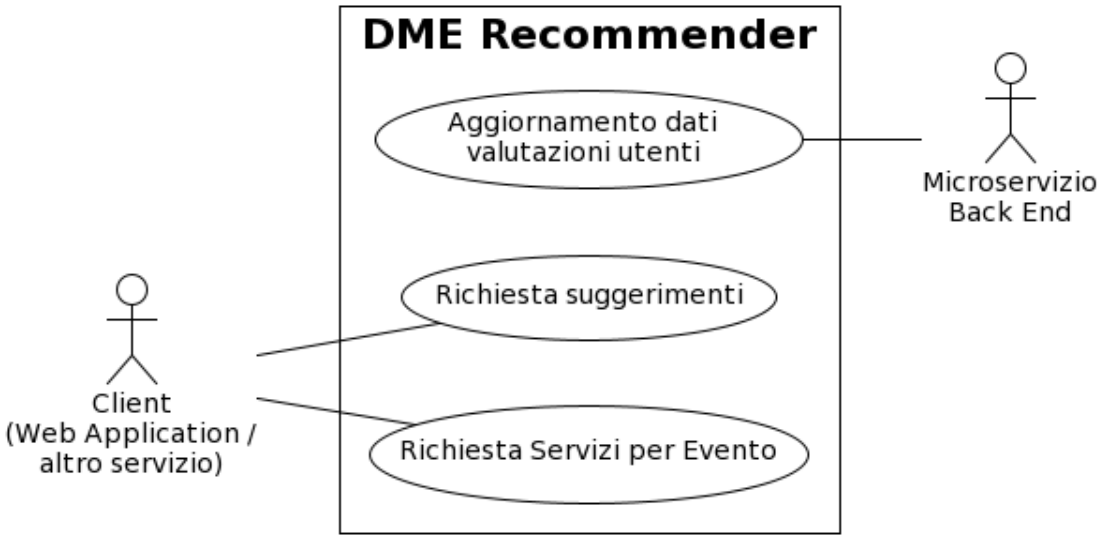

Figura 5.1. UML Use Case Diagram

Nome use case	Aggiornamento dati valutazioni utenti
ID use case	UC-1
Precondizione/i	• I microservizi sono funzionanti e non ci sono problemi di rete • È trascorso il tempo prestabilito dall'ultimo controllo
Postcondizione	Le valutazioni degli utenti nel database e le strutture dati dell'algoritmo di raccomandazione sono aggiornate
Descrizione scenario nominale (sequenza di azioni)	1. Il sistema scarica dal servizio di back end le valutazioni a partire dal timestamp memorizzato. 2. Il sistema aggiunge al db i nuovi utenti e fornitori. 3. Il sistema aggiunge al db le nuove valutazioni 4. Il sistema aggiorna le matrici dell'algoritmo di raccomandazione nel db. 5. Il sistema memorizza il timestamp della richiesta.
Altri scenari non nominali (varianti del nominale)	1.a Non ci sono nuovi dati. La procedura termina.

Tabella 5.1. Use Case 1

Nome use case	Richiesta suggerimenti
ID use case	UC-2
Precondizione/i	Il client dispone di un token di autenticazione valido.
Postcondizione	Il client ha ricevuto i suggerimenti richiesti
Descrizione scenario nominale (sequenza di azioni)	1. Il client richiede suggerimenti su fornitori. 2. Il sistema risponde con i suggerimenti per l'utente autenticato.
Altri scenari non nominali (varianti del nominale)	1.a Il client specifica nella richiesta dei parametri di filtraggio. 1.a Il sistema risponde con i suggerimenti per l'utente, filtrati secondo i parametri specificati dal client.

Tabella 5.2. Use Case 2

Nome use case	Richiesta Servizi per Evento
ID use case	UC-3
Precondizione/i	Il client dispone di un token di autenticazione valido.
Postcondizione	Il client ha ricevuto le informazioni richieste
Descrizione scenario nominale (sequenza di azioni)	1. Il client richiede le tipologie di servizi associate ad une evento. 2. Il sistema risponde con le informazioni richieste.
Altri scenari non nominali (varianti del nominale)	

Tabella 5.3. Use Case 3

5.2 Stato dell'arte e tecnologie utilizzate

L'utilizzo dell'architettura a microservizi ha permesso la scelta autonoma dei framework e del DBMS dal resto dell'applicazione. Di seguito sono descritte le tecnologie impiegate per la realizzazione del microservizio.

5.2.1 DBMS a grafo Neo4j

Un DBMS a grafo è un DBMS di tipo NoSQL che utilizza i grafi come strutture per memorizzare le informazioni. Le entità sono rappresentate dai nodi del grafo, mentre le relazioni tra loro sono rappresentate dagli archi che collegano i nodi.

Questo tipo di DBMS mette al centro le relazioni, trattandole con la stessa importanza delle entità. Le relazioni sono memorizzate insieme alle entità che esse collegano e ciò comporta un notevole miglioramento prestazionale nelle interrogazioni che attraversano milioni di relazioni, rispetto ai DBMS relazionali usati tradizionalmente. Questi ultimi, infatti, richiedono l'esecuzione di operazioni di JOIN tra tabelle in fase di interrogazione che possono essere molto dispendiose. I database a grafo, invece, consentono operazioni sulle relazioni con tempo costante. Grazie a questi vantaggi prestazionali, processi che venivano eseguiti in batch ora sono realizzabili in tempo reale.

I DBMS a grafo sono particolarmente adatti a quelle applicazioni in cui sono importanti le relazioni tra i dati, quindi che richiedono l'attraversamento di un elevato numero di relazioni in tempo reale. Alcuni di queste sono:

- *Social network*, dove contano le relazioni tra gli utenti, ad esempio la ricerca di amici di amici di un utente.
- *Rilevamento di frodi*, dove è importante analizzare in tempo reale relazioni tra transazioni finanziarie, ad esempio quelle che usano la stessa carta di credito.
- *Real-Time Recommender System*, grazie alla possibilità di analizzare in tempo reale relazioni tra informazioni come le preferenze degli utenti, la cronologia degli acquisti, le valutazioni espresse, ecc.

Modellazione dei dati

In un database a grafo, uno dei principali modelli di utilizzati per rappresentare i dati è quello del *Grafo di Proprietà*, costituito da nodi, relazioni e proprietà.

I nodi del grafo rappresentano le entità e possono avere un'etichetta che ne indichi la categoria di appartenenza. Inoltre ad un nodo possono essere assegnate molteplici proprietà, nella forma di attributi chiave-valore. Le relazioni tra nodi sono rappresentate dagli archi del grafo. Queste hanno un significato semantico che è indicato dal tipo della relazione. Le relazioni sono sempre direzionali, cioè vanno da un specifico nodo di origine ad uno di destinazione, ma possono essere attraversate anche nel verso opposto. Inoltre, come i nodi, anche le relazioni possono avere delle proprietà. Tra due nodi possono esserci più relazioni.

Figura 5.2. Esempio di grafo di proprietà

Nel grafo di proprietà in figura 5.2 sono presenti due nodi, etichettati come *Person* e *Car*. Entrambi hanno una serie di proprietà definite con coppie chiave valore. Ad esempio, il nodo di tipo *Person* ha la proprietà *first_name* il cui valore è "Umberto". Il nodo *Person* ha una relazione uscente verso il nodo *Car*, di tipo *OWNS*. Anche la relazione ha una proprietà: *since*.

Questo modello viene spesso considerato "whiteboard-friendly" poiché la rappresentazione dei dati è simile a quella che si può disegnare su una lavagna quando si definisce il dominio dei dati di un'applicazione, quindi particolarmente intuitiva. La trasposizione dal modello fisico disegnabile su una lavagna allo schema del database consiste nel definire etichette e proprietà, ma senza stravolgere la struttura del modello. Ciò è in contrapposizione al modello relazionale, che prevede la rappresentazione dei dati in tabelle normalizzate, perciò lontana da quella intuitiva che si abbozza su una lavagna.

Neo4j

Neo4j è un DBMS a grafo open-source, scritto in Java e Scala, sviluppato da Neo4j, Inc., disponibile pubblicamente dal 2007. Secondo la classifica DB-Engines, Neo4j è il diciannovesimo database engine per popolarità ed è il primo nella categoria dei DBMS a grafo.

Neo4j è basato sul modello del Grafo di Proprietà e lo utilizza anche a livello di archiviazione, cioè fa uso di puntatori tra i dati per consentire la navigazione tra le relazioni a tempo costante. Un ulteriore vantaggio è l'agilità, cioè la capacità di adattare facilmente lo schema all'evoluzione del dominio dei dati, senza richiedere migrazioni dei dati esistenti.

Neo4j supporta le transazioni ACID e molte delle caratteristiche presenti in altre tipologie di DBMS. Neo4j supporta il linguaggio Cypher: un linguaggio di interrogazione simile ad SQL, ma per interrogare database basati sui grafi, supportato anche da DBMS a grafo diversi da Neo4j.

Infine, sono presenti driver e librerie di supporto per i principali linguaggi di programmazione.

Cypher Query Language

Cypher è un linguaggio dichiarativo, espressamente concepito per l'interrogazione e l'aggiornamento di database a grafo. Prende ispirazione da SQL, linguaggio dichiarativo utilizzato con i database relazionali, e fa uso di una sintassi basata su ASCII-Art che lo rende molto intuitivo. Una query Cypher descrive, attraverso questa sintassi, il pattern che descrive i nodi e le relazioni desiderati. I nodi sono rappresentati da una coppia di parentesi tonde, mentre le relazioni si indicano tramite frecce.

Facendo di nuovo riferimento al grafo di figura 5.2, la query per ottenere tutte le persone che possiedono un'automobile di marca "FIAT" è la seguente:

```
MATCH (p:Person)-[:OWNS]->(:Car {maker: "FIAT"})
RETURN p
```

Nella clausola MATCH si chiede di selezionare tutti i nodi etichettati *Person* che sono legati a nodi etichettati *Car*, attraverso una relazione di tipo *OWNS* uscente, indicandoli con la variabile *p*. Nella clausola RETURN si chiede quindi di restituire come risultato i nodi indicati dalla variabile *p* nella clausola MATCH.

Altre clausole sono presenti per effettuare diverse operazioni, come DELETE, CREATE e SET.

5.2.2 Spring

Spring è un framework per la creazione di applicazioni Java (oltre che Groovy e Kotlin) in ambito enterprise, sviluppato da Pivotal Software.

Nella sua versione di base, Spring non impone scelte relative all'architettura e all'organizzazione del codice, ma si limita ad offrire un modello di configurazione e un meccanismo di dependency injection. Questa caratteristica lo rende flessibile al punto da poter essere adoperato in una moltitudine di scenari, dalle applicazioni embedded al cloud. Infatti, alla base della filosofia di Spring, c'è la volontà di fornire massima libertà di scelta a tutti i livelli, senza essere opinionato.

In aggiunta al modulo core, Spring offre un'ampia gamma di moduli da utilizzare per scenari specifici.

Spring implementa un principio chiamato *Inversion of Control*. In generale, questo stabilisce che invece di essere il codice dell'applicazione ad invocare funzioni offerte dalla libreria, è il framework ad invocare il codice dell'applicazione, quindi a gestire il flusso di esecuzione dell'applicazione. Spring applica questo principio attraverso la *Dependency Injection*: gli oggetti si limitano a definire le proprie dipendenze, ad esempio attraverso parametri dei costruttori, mentre il framework si occupa di gestire questi oggetti e iniettarli in fase di creazione dell'oggetto. Gli oggetti il cui ciclo di vita è controllato da Spring prendono il nome di *bean*.

La Dependency Injection in Spring avviene attraverso un componente del framework chiamato *Inversion of Control Container*. Il container ha il compito di istanziare, configurare e assemblare i bean richiesti dalla configurazione dell'applicazione. Ci sono vari modi per definire la configurazione, il più comune è quello di apporre agli oggetti delle annotazioni Java.

Grazie alla Dependency Injection, il codice diventa molto più pulito. I bean non devono provvedere ad istanziare o a procurarsi istanze esistenti dei bean da cui dipendono. Inoltre questo meccanismo consente un disaccoppiamento tra bean e dipendenze che comporta migliore testabilità, poiché ciascun bean non è legato ad una particolare istanza della propria dipendenza. Ciò si può realizzare indicando come dipendenze classi astratte o interfacce, quindi sostituibili con implementazioni "mock" o "stub".

Le dipendenze di un bean possono essere indicate attraverso i parametri del costruttore o di un setter. I bean sono singleton, quindi se più bean dipendono dallo stesso tipo di bean, questi ricevono la stessa istanza. Inoltre la creazione di un bean da parte del container avviene solo se questo bean è richiesto. Molti di questi comportamenti sono configurabili in modo diverso, se ritenuto opportuno.

Un'altra caratteristica offerta dal framework Spring è la programmazione orientata agli aspetti (*Aspect Oriented Programming*). Un aspetto è un comportamento che riguarda più classi (*cross-cutting concern*), per cui il relativo

codice andrebbe replicato all'interno di ogni classe interessata. Ad esempio, ogni metodo che interagisce con un database può essere interessato ad usare le transazioni, quindi deve occuparsi delle operazioni di gestione (inizio, commit e rollback in caso di errori). La programmazione orientata agli aspetti consente di definire queste operazioni in modo da poterle riutilizzare ovunque sia necessario. Quindi, prima e/o dopo l'esecuzione di un metodo si può eseguire l'aspetto.

Spring Boot

Spring Boot è uno strumento della famiglia di Spring che consente la creazione rapida di applicazioni basate sul framework Spring, con una serie di configurazioni prestabilite, denominate *starter*. A differenza del framework Spring di base, che tende a massimizzare la flessibilità non imponendo scelte, Spring Boot è opinionato. Tuttavia, in base alle esigenze, le configurazioni di default possono essere sovrascritte. Questo consente di iniziare lo sviluppo dell'applicazione senza doversi preoccupare di definire ogni singolo aspetto della configurazione e, successivamente, in modo graduale, apportare delle modifiche al variare dei requisiti.

Il meccanismo di autoconfigurazione è concepito in modo da intervenire solo quando non sono state esplicitate delle configurazioni manuali. Tali configurazioni possono essere definite tramite classi di configurazione, appositamente annotate con *@Configuration*.

Spring Boot fornisce anche componenti come container Servlet embedded, per cui non è necessario configurarne uno proprio.

Tra i vari starter offerti da Spring Boot, c'è *spring-boot-starter-web* che consente la creazione di un'applicazione Spring Web MVC, dotata di servlet container embedded Tomcat.

Spring Web MVC

Spring Web MVC è un framework basato su Spring e sull'API Servlet, per la creazione di applicazioni web secondo il pattern Model View Controller.

L'architettura di un'applicazione Spring Web MVC prevede l'esistenza di un unico Servlet che riceve tutte le richieste, chiamato *DispatcherServlet*. Questo, però, delega le operazioni di gestione a componenti configurabili, quindi vi è un'astrazione sopra l'API Servlet.

I componenti di un'applicazione Spring Web MVC sono definiti come bean Spring, annotati come *@Component*, in modo da essere iniettabili attraverso

il container di Spring. Questa annotazione generica è spesso sostituita da annotazioni che definiscono uno specifico tipo di componente:

- *@Controller*: componente che gestisce le richieste dei client. I metodi di un controller vengono mappati alle richieste specificando URL e verbo HTTP attraverso apposite annotazioni e restituiscono un riferimento ad una risorsa, Nel caso di applicazioni web tradizionali, questa è una vista, mentre nella variante *@RestController* il dato restituito viene serializzato, ad esempio in formato JSON. Questa variante, infatti, si usa per servire API, invece di pagine web.
- *@Repository*: componente che consente operazioni CRUD su un database, secondo il pattern *DAO* (Data Access Object). Generalmente, se si usa il modulo Spring Data, invece di definire un componente repository, si usano interfacce fornite da Spring Data stesso.
- *@Service*: componente usato per la logica di business dell'applicazione.

Il pattern MVC prevede che il modello comprenda i dati e la logica dell'applicazione, la vista sia una rappresentazione dei dati e il Controller sia responsabile di ricevere le richieste, interagire con il modello e rispondere con una vista. In modo simile, in un'applicazione Spring Web, i controller gestiscono le richieste interagendo con i servizi, che a loro volta possono effettuare operazioni sui dati attraverso i repository, e rispondono con una vista.

Spring Data Neo4j

Spring Data comprende una serie di moduli spring che offrono un'interfaccia unificata per l'accesso ai dati, indipendentemente dalla tecnologia di persistenza che si usa. Infatti, Spring Data offre moduli per il supporto di svariate tipologie database relazionali o database NoSQL. Questi moduli offrono interfacce repository per l'accesso ai dati e supportano la mappatura dei dati in oggetti entità (*Object-Relational Mapping* nel caso di database relazionali).

Le interfacce repository consentono di definire query personalizzate a partire dai nomi dei metodi. Seguendo le giuste convenzioni sui nomi, infatti, è possibile estendere le interfacce repository aggiungendo dei propri metodi la cui implementazione viene generata automaticamente a partire dal nome e dalla signature. È inoltre possibile specificare manualmente il codice della query da associare ad un metodo, attraverso l'annotazione *@Query*.

Neo4j offre il modulo Spring Data Neo4j, basato sul driver Java Neo4j, che consente una facile integrazione tra le applicazioni Spring e i database Neo4j.

Nello specifico, è possibile implementare l'interfaccia *Neo4jRepository* o l'equivalente versione reactive. È inoltre possibile usare le interfacce agnostiche *Repository* e *CrudRepository*, offerte da Spring Data. In questo modo si perdono metodi specifici di Neo4j, ma si guadagna in flessibilità.

Le classi che rappresentano le entità vengono annotate con *@Node*, eventualmente specificando l'etichetta Neo4j da assegnare al nodo nel database. Queste hanno un campo annotato *@Id* che rappresenta la chiave e può essere autogenerato con *@GeneratedValue*.

In un'entità, una relazione con un'altra entità è annotata con *@Relationship*, che consente di specificare la direzione e il tipo. Per assegnare proprietà anche alla relazione tra due entità, nella classe del nodo di origine non si fa riferimento direttamente alla classe del nodo di destinazione, ma ad una classe che rappresenta la relazione. Questa va annotata con *@RelationshipProperties* e, oltre ai campi che indicano le proprietà, va messo un campo con il riferimento al nodo di destinazione, annotato con *@TargetNode*.

5.2.3 Altro

Altre tecnologie utilizzate:

- *Keycloak* è una soluzione di gestione degli accessi centralizzata (*Single Sign-on*) per applicazioni. L'accesso centralizzato è particolarmente importante nelle architetture a microservizi, dove ciascun servizio è un'applicazione a se stante. Keycloak supporta la registrazione, l'autenticazione e l'autorizzazione degli utenti. Per l'integrazione con le applicazioni, Keycloak offre una serie di Adapter per i principali linguaggi/framework, tra cui l'adapter per il modulo Spring Security.

- *Lombok* è una libreria Java che fornisce una serie di annotazioni che, in fase di compilazione, vengono processate e il relativo codice viene aggiunto al codice dell'applicazione. Ad esempio, le annotazioni Lombok *@Getter* e *@Setter* comportano la generazione dei metodi getter e setter di default per i campi a cui sono state apposte. Pertanto, il vantaggio che si ricava è la riduzione di gran parte del codice "boilerplate" e, di conseguenza, un codice più pulito.

- *Simple Logging Facade for Java (SLF4J)*: interfaccia comune per diversi framework di logging.

- *Maven*: strumento per la gestione dei progetti Java. Prevede la creazione di un file "pom.xml" nel quale vanno indicati i metadati e le dipendenze dell'applicazione. Nel caso di Spring Boot, questo viene generato automaticamente e può essere modificato successivamente. Offre diversi comandi per la compilazione, testing e impacchettamento delle applicazioni.

- *Tomcat*: container di servlet Jakarta (Java EE). Si tratta di un server web basato sull'API Servlet. Spring Boot, nella configurazione di default, prevede l'uso di un'istanza di Tomcat embedded.

- *Docker*: software per la gestione e l'esecuzione di container (virtualizzazione a livello del sistema operativo). Un container è un ambiente isolato per l'esecuzione di applicazioni e l'accesso alle risorse di sistema (reti, file system, ecc.) avviene in modo controllato. Rispetto al tradizionale approccio alla virtualizzazione, l'overhead in termini di prestazioni e uso di risorse è notevolmente ridotto. L'istanza di Neo4j utilizzata per lo sviluppo è un container creato a partire dall'immagine ufficiale.

- *RestTemplate*: API di Spring Web per eseguire richieste HTTP sincrone.

5.3 Modellazione dei dati

Il servizio riceve le valutazioni espresse dagli utenti nei confronti dei fornitori. Queste transazioni vengono modellate, nel grafo di proprietà come rappresentato in figura 5.3.

Figura 5.3. Grafo *User-Supplier*

Gli utenti sono nodi con etichetta *User* e unica proprietà *id*. I fornitori sono nodi con etichetta *Supplier* e proprietà *id*, *codGeo* e *codCategoria*. Queste ultime due proprietà rappresentano informazioni da usare per il filtraggio dei suggerimenti. Tra utenti e fornitori vi è una relazione di tipo *HAS_RATED*, la cui unica proprietà *value* rappresenta il voto assegnato dall'utente al fornitore.

Per l'implementazione dell'algoritmo Slope One, serve avere a disposizione, per ogni coppia di fornitori, la deviazione media tra due fornitori e il numero di utenti che hanno valutato entrambi i fornitori (cardinalità). Queste informazioni possono essere rappresentate in due matrici che, nel grafo delle proprietà, corrispondono a due relazioni per ciascun verso.

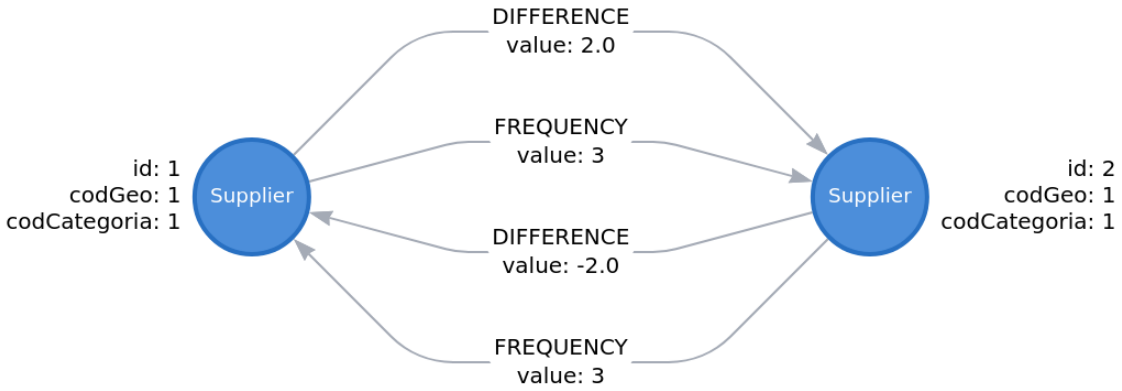

Figura 5.4. Grafo *Supplier-Supplier*

In figura 5.4 sono rappresentate le relazioni *DIFFERENCE* e *FREQUENCY* tra due nodi *Supplier*, con le rispettive proprietà *value*, in entrambi i versi. (Si sarebbe potuto anche usare una singola relazione per verso, con due proprietà, ma si è scelto di procedere in questo modo per la maggiore intuitività).

In uno sviluppo futuro, il microservizio dovrà fornire suggerimenti agli utenti sui tipi servizi da acquistare, in base al tipo di evento che si sta organizzando. In questa prima implementazione, questa funzionalità è stata implementata come prototipo, senza attingere alla cronologia delle transazioni, ma fornendo suggerimenti a partire da informazioni statiche presenti nel database. Ciò è stato realizzato attraverso due tipologie di nodi, *EventType* e *ServiceType*, e una relazione dal primo al secondo, *PROPOSES*.

Figura 5.5. Grafo *EventType-ServiceType*

5.4 Struttura del micorservizio

Il microservizio è composto da una serie di classi e interfacce che corrispondono a componenti Spring illustrate nei paragrafi precedenti. Di seguito è riportata una panoramica dei componenti realizzati.

Controller

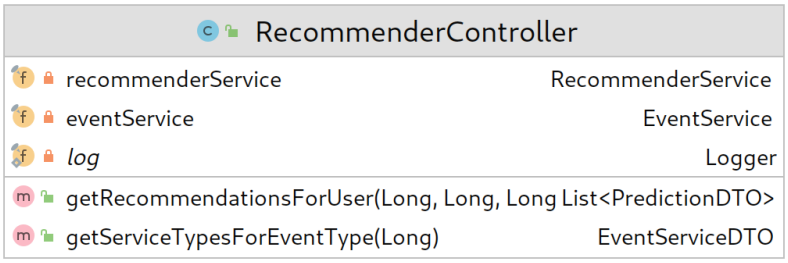

Figura 5.6. UML Class Diagram - Controller

I controller sono i componenti responsabili della gestione delle richieste da parte dei client, Nel caso di questo microservizio di back-end, le risposte non sono pagine web, ma dati serializzati in formato JSON, quindi si tratta di RestController.

L'unico controller realizzato è *RecommenderController*. Questo ha il compito di gestire entrambe le richieste offerte dall'API, ciascuna attraverso un metodo. La prima serve a richiedere suggerimenti di fornitori per un utente e consente di specificare come parametri opzionali *odCategoria* e *codGeo*. La seconda serve a richiedere le tipologie di servizio associate ad una tipologia di evento. Entrambe le richieste sono di tipo GET.

Servizi

I servizi implementano la logica dell'applicazione e possono essere interrogati dai controller per rispondere alle richieste.

- *ImporterService* (fig. 5.7) gestisce l'importazione delle valutazioni dal servizio di backend apposito. Questo offre una API in grado di restituire tutte le valutazioni rilasciate dagli utenti a partire da un certo momento

(timestamp). ImporterService, quindi, ha il compito di interrogare periodicamente questa API per ottenere nuove valutazioni (eventualmente insieme a nuovi utenti e fornitori), memorizzarle nel database e invocare i metodi offerti da RecommenderService per aggiornare le matrici dell'algoritmo Slope One. Tale operazione avviene ad intervalli regolari, attraverso la funzionalità di task scheduling offerta da Spring. L'intervallo di tempo è definito da un parametro regolabile.
Il servizio memorizza il timestamp di una richiesta in modo da usarlo per la richiesta successiva. L'interrogazione delle API REST di altri servizi avviene sfruttando l'API RestTemplate di Spring, che consente al microservizio di comportarsi da client HTTP.

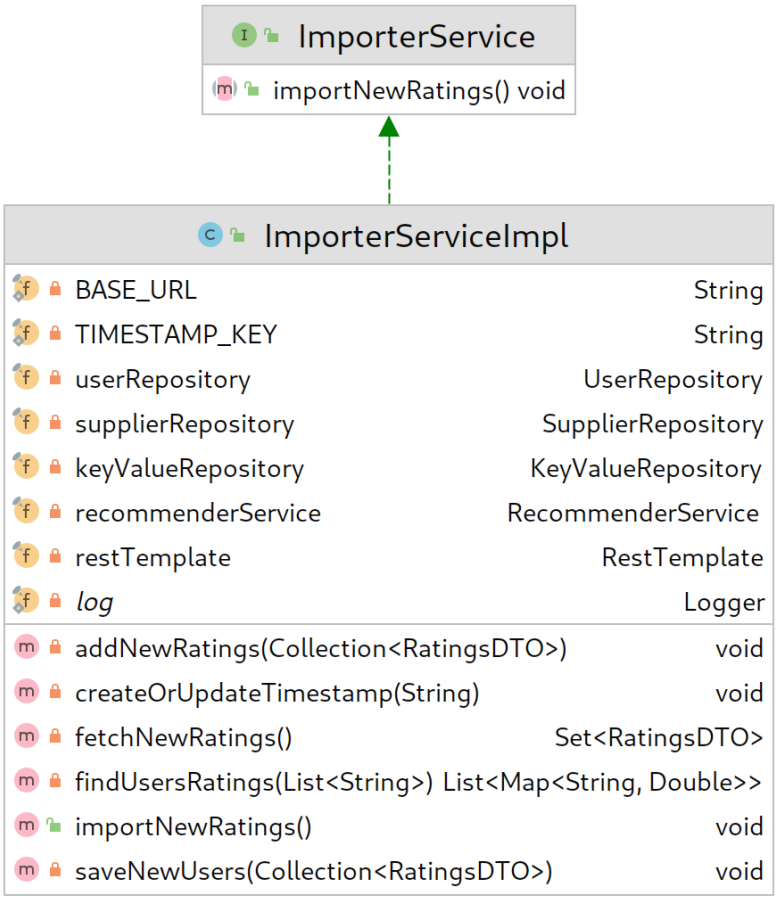

Figura 5.7. UML Class Diagram - ImporterService

- *RecommenderService* (fig. 5.8) è il servizio che si occupa di eseguire l'algoritmo SlopeOne per ottenere suggerimenti, oltre ad avere il compito di aggiornare le strutture dati necessarie al funzionamento dell'algoritmo, quando ImporterService importa nuove valutazioni. I suggerimenti generati vengono successivamente, se richiesto, filtrati in base a *codGeo* e *codCategoria*.

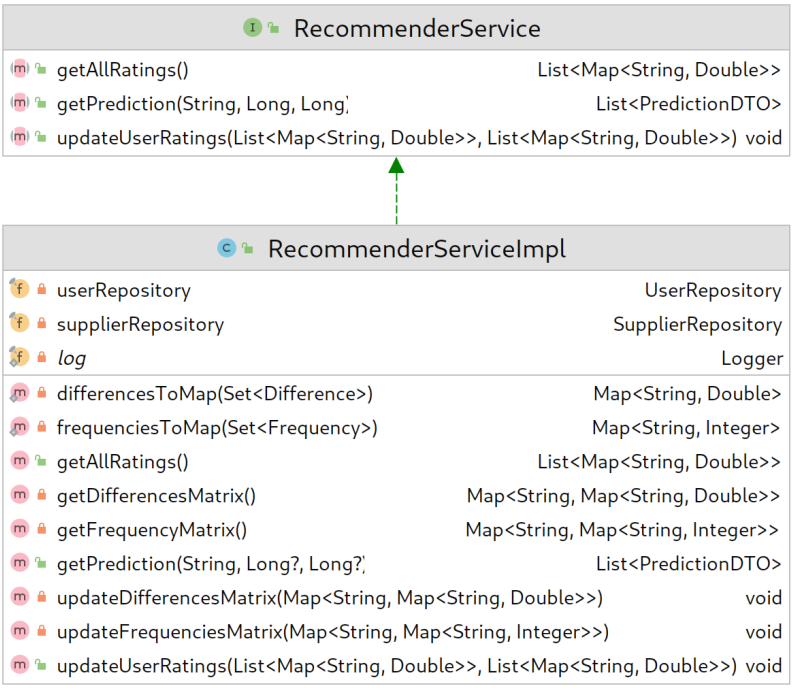

Figura 5.8. UML Class Diagram - RecommenderService

- *EventService* (fig. 5.9) implementa la logica che consente di importare nel database le associazioni tra tipologia di evento e tipologie di servizi da file JSON e la logica per interrogare il database e ottenere queste associazioni.

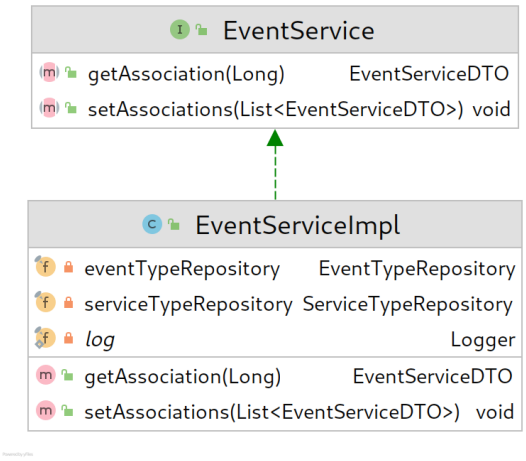

Figura 5.9. UML Class Diagram - EventService

Repository e Entità

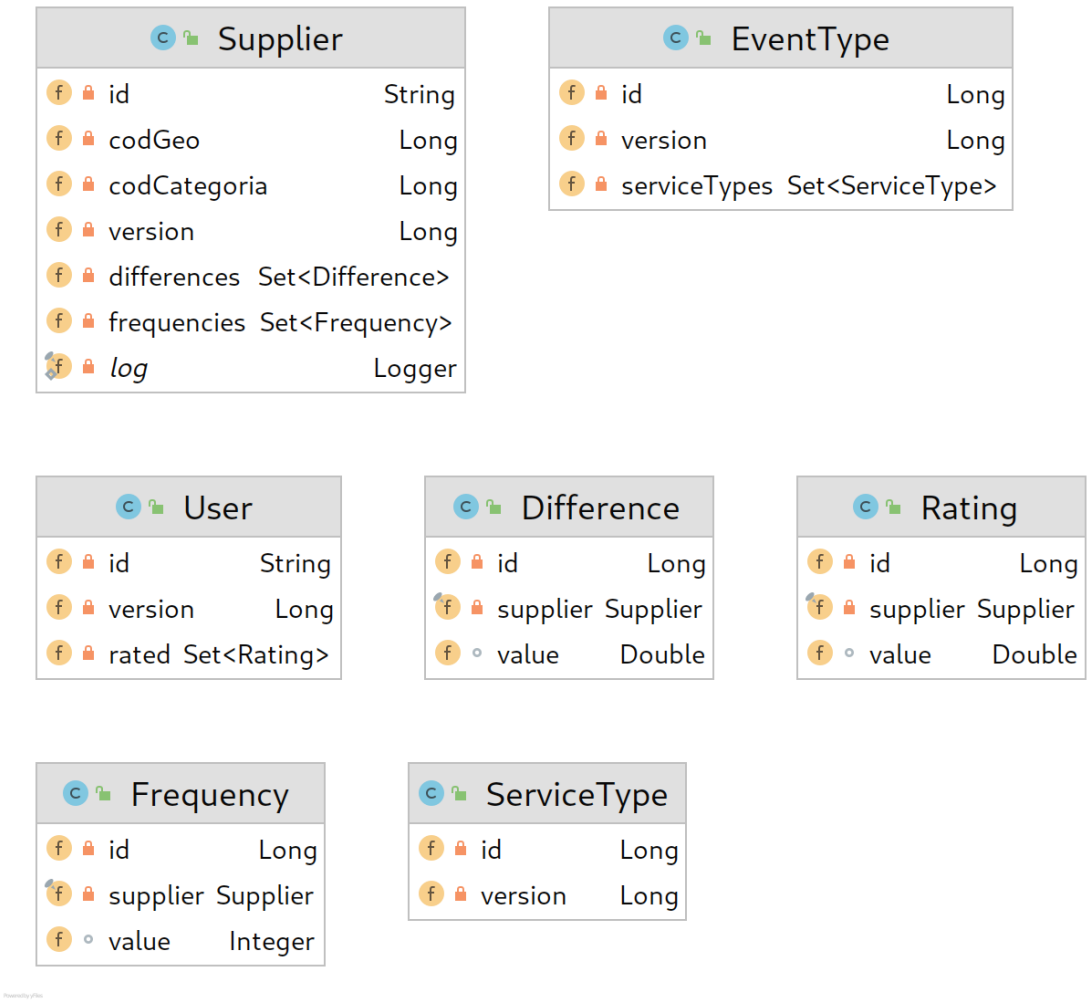

Figura 5.10. UML Class Diagram - Entità

Le entità sono classi corrispondenti alle tipologie di nodo identificate nel processo di modellazione dei dati. In aggiunta a queste, ci sono classi che rappresentano le relazioni dotate di attributi, come illustrato nel paragrafo relativo a Spring Data Neo4j.

Per ogni entità è stata definita un'interfaccia repository estendendo Neo4jRepository e specificando il tipo di entità e il tipo di Id.

I servizi usano i repository per interrogare lo strato di persistenza e ottengono i risultati sotto forma di oggetti entità.

DTO

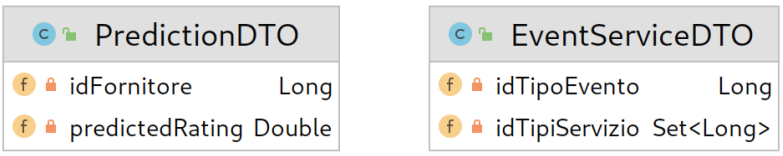

Figura 5.11. UML Class Diagram - DTO

I DTO sono classi che rappresentano i dati scambiati. Ci sono DTO che modellano le risposte ricevute dal servizio di back end, quindi modellati in base all'API che questo definisce. Inoltre, ci sono DTO che rappresentano le risposte che il microservizio recommender invia ai client.

L'uso di classi DTO diverse dalle classi entità consente il disaccoppiamento tra la rappresentazione interna dei dati e quella che si vuole offrire all'esterno. Quindi, i metodi pubblici dei servizi, generalmente interrogati dai metodi dei controller, restituiscono risultati attraverso oggetti DTO.

5.5 Testing

I metodi dei servizi, che implementano la logica di business dell'applicazione, sono stati testati tramite Unit Test, grazie alle funzionalità di testing integrate nel framework Spring.

Per quanto concerne la persistenza dei dati nel database Neo4j, ulteriori verifiche sono state effettuate tramite query manuali, scritte in linguaggio Cypher ed eseguite tramite l'interfaccia web offerta da Neo4j. Questo strumento consente di visualizzare i risultati delle query in vari formati tra i quali vi è anche il grafo di proprietà, che risulta particolarmente intuitivo.

Riguardo l'algoritmo di raccomandazione, l'obiettivo dei test è stato quello di determinare la correttezza dell'implementazione, piuttosto che l'accuratezza dei suggerimenti. A questo proposito, sono stati eseguiti test con dati di esempio e confrontati i risultati con implementazioni open source, in vari linguaggi di programmazione, dello stesso algoritmo. I valori ottenuti con il dataset di test sono risultati in linea con quelli prodotti dalle altre implementazioni testate.

I test di integrazione sono stati eseguiti parzialmente, a causa dello stato di avanzamento dello sviluppo del resto dell'applicazione. Il servizio di back

end da cui recuperare le valutazioni degli utenti è stato inizialmente simulato con un server HTTP REST locale e, successivamente, quando possibile, con un'istanza del servizio di backend. I metodi del controller sono stati testati con un client HTTP.

Capitolo 6

Conclusioni

6.1 Percorso svolto

Durante lo svolgimento dei lavori, sono state acquisite informazioni sulle architetture a microservizi, con i relativi vantaggi, problematiche e potenziali soluzioni. Si è altresì studiato il mondo dei Recommender System, i relativi usi e le varie tipologie, con particolare enfasi sulla categoria del Collaborative Filtering.

Successivamente, si è provveduto a realizzare un microservizio recommender da integrare nell'applicazione DME, per l'organizzazione e la gestione digitale degli eventi. Tale microservizio implementa l'algoritmo di collaborative filtering Weighted Slope One per suggerire fornitori di servizi agli utenti, in base ai propri gusti. L'algoritmo è stato selezionato per via del buon rapporto tra semplicità di implementazione e prestazioni che lo caratterizza.

Si è scelto di utilizzare un database a grafo, Neo4j, per i vantaggi prestazionali derivanti dalla caratteristica di considerare le relazioni tra i dati con la stessa importanza dei dati stessi, che ben si presta a questo tipo di applicazioni. Per la parte applicativa si è utilizzato il framework Java Spring, con i relativi moduli.

I test effettuati confrontando i risultati dell'esecuzione dell'algoritmo di raccomandazione con quelli forniti da altre implementazioni open source dello stesso algoritmo, a parità di dati di input, ne hanno confermato la corretta implementazione.

A causa delle tempistiche di sviluppo di altri componenti dell'applicazione, è stata realizzata solo una parte dei test di integrazione. Il deployment e i rimanenti test di integrazione con il resto dell'applicazione avverranno nelle prossime settimane.

6.2 Sviluppi futuri

Il microservizio realizzato rappresenta un punto di partenza per fornire un'esperienza personalizzata agli utenti della piattaforma. Uno dei primi sviluppi sarà quello di terminare l'implementazione della funzionalità di suggerimento di tipologie di servizi in base all'evento che si sta organizzando. Ciò è stato realizzato in via prototipale e dovrà essere portato a termine.

L'utilizzo dell'architettura a microservizi, grazie alla flessibilità offerta, consente di apportare modifiche implementative in modo indipendente dal resto dell'applicazione. A patto di coordinare con gli altri servizi eventuali modifiche all'interfaccia di programmazione esposta, se ritenuto opportuno si può anche decidere di cambiare totalmente le tecnologie, senza impatto sugli altri servizi.

L'algoritmo di Collaborative Filtering utilizzato può essere successivamente migliorato con strumenti più sofisticati, ad esempio l'uso di tecniche model-based, basate su machine learning e le tecniche di raccomandazione ibride, che sfruttano informazioni sulle caratteristiche degli oggetti da consigliare. In particolar modo quando saranno presenti dati su transazioni reali degli utenti, sarà possibile studiare un modello di suggerimento più adatto alle esigenze dell'applicazione.

Specialmente in ottica futura, con l'impiego di tecniche di raccomandazione più avanzate, è ipotizzabile che la scelta di utilizzare un database a grafo risulti vantaggiosa, essendo i Recommender System una delle principali applicazioni di questa tipologia di database.

Bibliografia

[1] Francesco Ricci, Lior Rokach, Bracha Shapira, *Recommender Systems Handbook*, Springer Science+Business Media, New York, 2015, 2nd ed.

[2] Jannach, D., Zanker, M., Felfernig, A., Friedrich, G., *Recommender Systems: An Introduction*, Cambridge: Cambridge University Press, 2010.

[3] Daniel Lemire, Anna Maclachlan, *Slope One Predictors for Online Rating-Based Collaborative Filtering*, In SIAM Data Mining (SDM'05), Newport Beach, California, 2005.

[4] Chris Richardson, *Microservice Architecture*, https://microservices.io/.

[5] Chris Richardson, Floyd Smith, *Microservices From Design to Deployment*, NGINX.

[6] Dmitry Namiot, Manfred Sneps-Sneppe, *On Micro-services Architecture*, International Journal of Open Information Technologies, 2014.

[7] *Formulario "Ricerca e Sviluppo" Progetto DME.*

[8] *Neo4j Developer Resources*, https://neo4j.com/developer.

[9] *DB-Engines Ranking*, https://db-engines.com/en/ranking.

[10] *Spring Framework Documentation*, https://docs.spring.io/spring-framework/docs/current/reference/html.

[11] *Spring Boot Reference Documentation*, https://docs.spring.io/spring-boot/docs/current/reference/html.

[12] Gerrit Meier, Michael Simons, *Spring Data Neo4j*, https://docs.spring.io/spring-data/neo4j/docs/current/reference/html.