Università di Catania

Progetto Ingegneria dei Sistemi Distribuiti A.A 24/25

Supermarket online

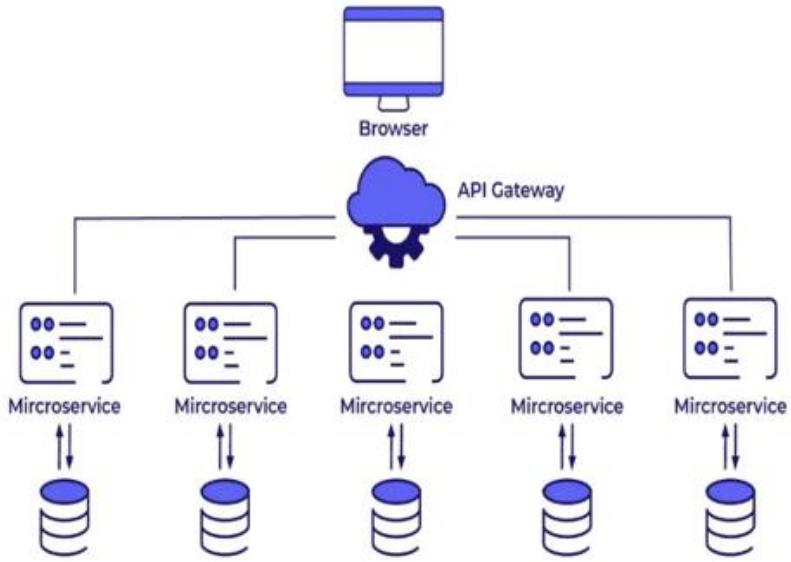
Realizzato dallo studente:

Vincenzo Barba

1000002746

Docente di riferimento:

prof. Emiliano Tramontana



Link repository pubblico:

https://github.com/enzobarba/IDS_Project

Sommario

1. Requisiti tecnici

Il seguente progetto ha come obiettivo quello di implementare un sistema software capace di elargire un servizio di supermercato online, progettato tramite un'architettura a microservizi e l'utilizzo dei minimi indispensabili design pattern.

In particolare, deve essere possibile gestire tre diversi tipi di utenze:

- Acquirenti, cioè i clienti che vogliono acquistare i prodotti;
- Rifornitori, coloro i quali hanno il compito di rendere disponibili online i prodotti effettivamente presenti nell'inventario reale;
- Admin, capaci di gestire eventuali problematiche tecniche all'interno del sistema, il che implica una concessione di permessi più elevati rispetto le altre categorie.

Perciò è necessario implementare un corretto e sicuro sistema di **autenticazione** e **autorizzazione**, affinché ogni utente possa eseguire esclusivamente le operazioni di cui necessita e per cui dispone dei relativi permessi, in base al ruolo assegnato all'atto di registrazione.

2. Strumenti utilizzati

Editor codice: VS Code (https://code.visualstudio.com/)

Ambiente di sviluppo: JDK 21 (https://www.oracle.com/it/java/technologies/downloads/)

Gestore progetto: Maven (https://maven.apache.org/download.cgi)

Framework per microservizi: SpringBoot (https://start.spring.io/)

Gestore container: Docker (https://www.docker.com/)

3. Soluzione proposta

L'intero sistema lato server è suddiviso in tre microservizi:

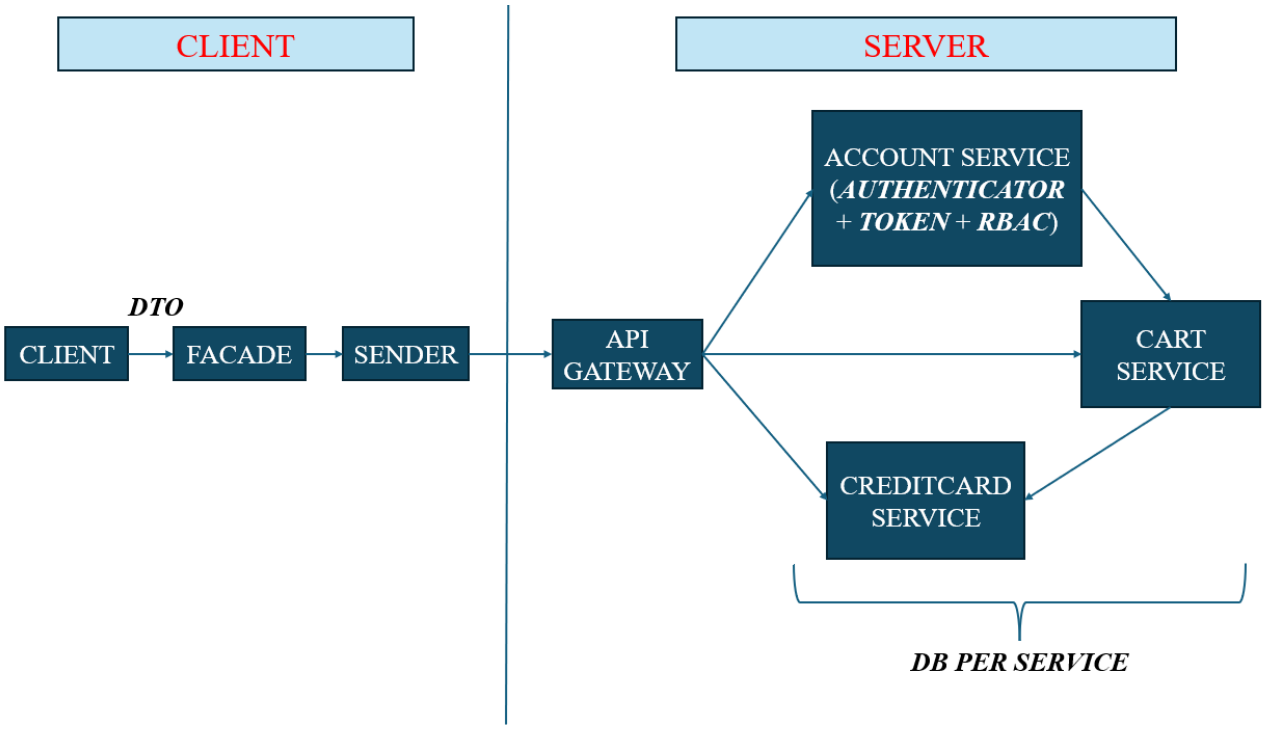
- Account Service, che permette di gestire le utenze in termini di registrazione, login, logout e concessione del ruolo;

- Product Service, tramite il quale è possibile manipolare l'inventario di prodotti messi a disposizione online, creandoli, eliminandoli, o rifornendo prodotti già esistenti. Inoltre, contiene tutta la logica dei carrelli, checkout e degli ordini effettuati;

- CreditCard Service, che simula un microservizio di pagamento, per mezzo del quale è possibile creare più carte di credito e assegnarle a uno specifico utente, con una certa quantità iniziale di denaro disponibile.

I microservizi non sono accessibili in modo diretto dall'esterno, in quanto le porte di ascolto dei rispettivi processi sono esposte solo all'interno della dockerizzazione. L'unico modo per il Client di interagire con l'intero sistema è quello di comunicare con l'API Gateway (la cui porta esposta all'esterno è la 8080, vedi file '*/docker-compose.yml'*), che gira su un altro container. Il client, senza interfaccia grafica, si limita a eseguire delle chiamate di test che coprono gran parte delle funzionalità e delle dinamiche possibili dell'intero software.
Le richieste inviate dal Client vengono incapsulate all'interno di appositi DTO, e vengono smistate all'API Gateway e ai vari microservizi (in pochi casi anche tra microservizi stessi), grazie a chiamate REST API, implementate in modo sincrono (per semplicità in ambiente di demo del progetto, implementabili facilmente in modo asincrono, vedi Capitolo 4).

Nella seguente figura viene mostrata la progettazione del sistema con le componenti principali:



Il design pattern **Authenticator** è implementato nel microservizio Account in quanto è necessario centralizzare il sistema di autenticazione, sia per garantire aspetti funzionali per l'utente, che soprattutto per motivi di sicurezza; infatti, si vuole verificare che l'utente che vuole fare login sia effettivamente chi dice di essere: il lato critico da proteggere è in questo caso l'aspetto economico, l'utilizzo di carte di credito.
Una volta richiesto il login, in caso di credenziali corrette, viene concesso un **token** al Client, che funge da prova di autenticazione passata, con una scadenza fissata a quattro ore. La durata è stata scelta considerando che non è un applicativo particolarmente critico e che, in media, l'utente inizia a riempire il carrello e a fare il checkout nel giro di mezza giornata circa, o anche meno. Ciò potrebbe sembrare un problema per l'usabilità e l'esperienza del cliente con l'applicativo, ma è un modo per abbassare la probabilità di attacchi, dato che manca un'autenticazione più forte, come 2FA, ritenuta però eccessiva; inoltre, considerando che è un applicativo con cui in media si interagisce un paio di volte al mese circa, reinserire le credenziali non dovrebbe essere motivo di scarsa usability.

Il design pattern **RBAC** (Role Based Access Control) permette, previa autenticazione, di concedere i permessi strettamente necessari (principio need to know) all'utente loggato, in base al ruolo attribuitogli in fase di registrazione. In

questo caso un permesso è rappresentato non dalla possibilità di accedere direttamente a un oggetto specifico, ma di effettuare una ristretta cerchia di chiamate ai microservizi tramite Api Gateway. È possibile consultare i permessi concessi ad ogni singolo ruolo nel file '*.../resources/db.prop*'.
Si è optato per questo design pattern, invece che del Reference Monitor, in quanto le utenze attese sono tante, in particolar modo per gli acquirenti, ma anche per i rifornitori. Inoltre, è anche abbastanza semplice individuare già in fase di progettazione le poche differenti tipologie di account; quindi, ha molto più senso attribuire i permessi ai ruoli, e di conseguenza mappare ogni utenza a un ruolo, invece che settare i permessi per ogni singolo profilo.
Si è scelto di non consentire agli utenti di poter avere più ruoli con un singolo account in quanto non ritenuto necessario. Così come l'Authenticator, esso è sviluppato nel microservizio Account, in quanto a stretto contatto col database dei profili.

Un'altra componente essenziale del sistema lato server è l'**API Gateway**. In un sistema distribuito è altamente consigliato ricevere le richieste dai Client da un unico punto di accesso, sia per questioni funzionali (astrae la complessità sottostante dei singoli microservizi), ma soprattutto per questioni di sicurezza, in quanto è questa componente a occuparsi di controllare autenticazione e autorizzazioni ad ogni richiesta ricevuta, per poi smistarla all'opportuno microservizio, se e solo se l'utente è stato autenticato e dispone dei permessi per far ciò.

Per trasferire i dati delle richieste dal client al server, vengono utilizzati appositi **DTO**, considerando scomodo il passaggio di multipli parametri per gran parte delle richieste, e in quanto non è ritenuto necessario far conoscere gli interi oggetti del dominio lato server al client. La serializzazione dei DTO avviene automaticamente in JSON nelle chiamate POST, tramite l'invocazione del metodo '*.bodyValue(DTO)'* sull'oggetto WebClient utilizzato per effettuare chiamate REST.

Ogni microservizio ha il suo database, mantenendo la linea del design pattern **Database per Service**. L'unico modo che un microservizio ha per accedere al DB di un altro microservizio, è quello di interrogare il controller del servizio in questione, senza un accesso diretto. Questo per avere un accoppiamento lasco tra i servizi e che le modifiche a un DB non abbiamo un impatto sugli altri.

Tutte le richieste che il client può effettuare si distinguono in due categorie differenti: la prima, di cui fanno parte quelle che non necessitano né di autenticazione né di autorizzazione (vedi Capitolo 4 per futura aggiunta della visione dei prodotti), di cui fanno parte la registrazione di un account e il tentativo di login, per ovvie ragioni funzionali;
la seconda comprende tutto il resto delle chiamate, che necessitano, tra gli altri parametri, un token valido: una volta ricevute dall'API Gateway, vengono prima di tutto indirizzate al microservizio Account grazie alla classe *'.../helper/RestHelper.java'*, che dispone un metodo per qualsiasi chiamata che necessiti un precedente controllo di autenticazione e permessi; se il servizio account risponde che l'utente ha un token valido (firma e timestamp) e l'autorizzazione per effettuare quella richiesta, allora verrà restituito l'username dell'utente associato, così che l'API Gateway smisti la chiamata all'opportuno microservizio (questi ultimi, escluso Account Service, non conoscono i token, ma hanno associazioni dei propri dati solo con username).

Tra le poche chiamate che i microservizi eseguono tra loro, vi è quella di Account Service, che durante la creazione di un account, invoca sul servizio Product la creazione di un carrello vuoto associato a quell'utenza, e quella del checkout del carrello, che chiede al servizio CreditCard di tentare il pagamento.
La chiamata a **checkout** è quella più articolata e ciò ne giustifica una sommaria descrizione:

1. Dopo aver appurato autenticazione e autorizzazioni, si cerca il carrello associato tramite l'username. Se esso è vuoto, o contiene almeno un prodotto la cui quantità non è più disponibile nell'inventario, allora viene restiuito un messaggio di errore.

2. Se questi controlli vanno a buon fine, si tenta di eseguire il pagamento tramite la chiamata al servizio esterno CreditCard, il quale controlla che la carta passata sia esistente, associata a quell'username, e che abbia abbastanza credito;

3. se anche questo controllo è ok, vengono decrementate le quantità dall'inventario, trasformato il carrello in ordine (congelato) e svuotato il carrello.

4. Possibili migliorie

- Inserire Audit e RateLimiter in APIGateway;
- Dare possibilità a utenti non autenticati di vedere i prodotti e creare un proprio carrello da mantenere in sessione tramite il design pattern SessionState;
- Implementare interfaccia grafica lato Client, e inserirvi controlli preliminari sulla coerenza dei dati inseriti nelle richieste;
- Gestire acquisti concorrenti in chiamata checkout;
- Utilizzare database su disco invece che database in memoria principale (impostazione attuale, tramite H2) per mantenere i dati anche allo spegnimento dei container;
- Implementare un metodo alternativo per far registrare un utente col ruolo desiderato rispetto al codice d'invito, o quanto meno nascondere il codice d'invito che è in chiaro nel codice, per questioni di sicurezza;
- Aggiungere daemon che periodicamente controlla l'insieme di token ed elimina quelli scaduti;
- Implementare il design pattern EventSourcing per garantire coerenza tra microservizi, auditabilità e una migliore scalabilità;
- Utilizzare chiamate asincrone invece che sincrone, facilmente modificabile in quanto WebClient le supporta già nativamente;
- Restituire oggetti più complessi in risposta al client invece che delle semplici stringhe;
- Gestire richieste malformate e conseguenti eccezioni lato server in modo più articolato e sicuro.