Università di Catania

Gamble with microservices

Ingegneria Sistemi Distribuiti e Laboratorio (LM-18)
Università degli Studi di Catania - A.A 2022/2023

Danilo Leocata - 1000022576
Marco Bongiovanni - W82000189
Docenti: Emiliano Alessio Tramontana, Andrea Fornaia

January 27, 2023

1 Introduzione al progetto

Il progetto presentato ha come obbiettivo la realizzazione del backend relativo ad un applicativo web che ha come scopo quello di fornire le API necessarie al corretto funzionamento di un sistema di gioco online, incentrato su eventi sportivi calcistici. Tra le funzionalità offerte dal servizio le principali comprendono:

- **Gestione utenti**: creazione, autenticazione e validazione delle richieste;
- **Gestione degli eventi** su cui gli utenti possono piazzare delle scommesse;
- **Gestione delle scommesse** degli utenti;

Per la realizzazione di tali funzionalità, è stato scelto di utilizzare un approccio a microservizi, ciò permetterà di sviluppare e distribuire i diversi componenti del nostro sistema in modo indipendente l'uno dall'altro, rendendo il sistema scalabile e meno oneroso da mantenere. Un' analisi del dominio ha portato alla realizzazione dei seguenti microservizi:

- `application-service`: espone le chiamate che saranno disponibili ai client per usufruire delle funzionalità offerte;
- `authentication-service`: componente non esposto ai client, si occupa della gestione dei dati utente e della relativa validazione;
- `game-service`: componente non esposto ai client, si occupa della gestione degli eventi sportivi;
- `session-service`: componente non esposto ai client, implementa le funzionalità necessarie alla gestione delle sessioni degli utenti.

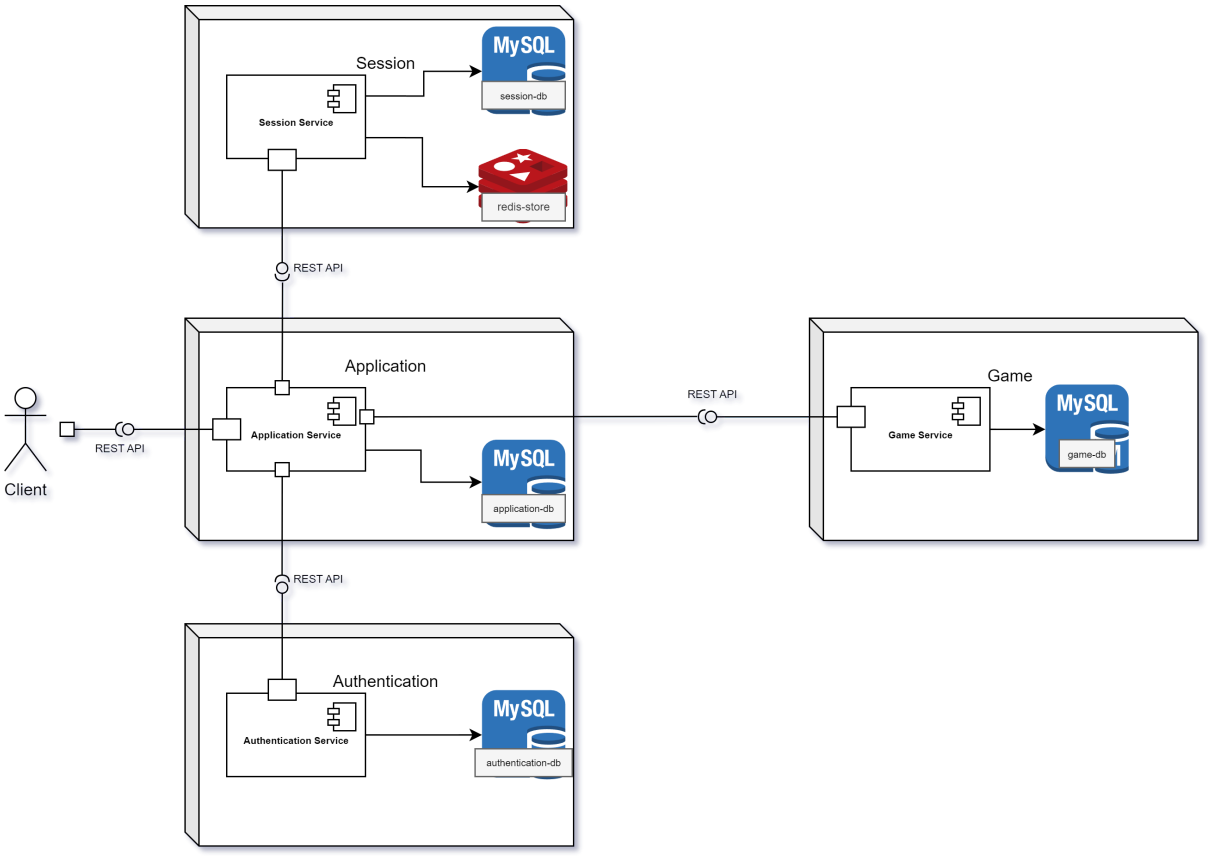

Figure 1: Architettura rete triplet

Il **codice sorgente** e la relativa **documentazione** di tali servizi è disponibile sulla piattaforma GitHub al seguente indirizzo: https://github.com/ingegneria-sistemi-distribuiti-2023. Si nota che non sono state implementate le funzionalità di: controllo eventuale schedina vincente e relativo pagamento, validità delle partite al momento della `place-bet` (con controllo di integrità della quota relativa alle partite), in quanto non ritenute necessarie ai fini della presentazione del progetto.

2 Requisiti funzionali

2.1 Application Service

Questo servizio ha come compito quello di rendere disponibile le API, necessarie ad un eventuale client, di interagire correttamente con la piattaforma di gioco, in dettaglio:

- Consentire a qualsiasi utente di effettuare una richiesta di registrazione alla piattaforma, tramite nome utente e password;
- Consentire a qualsiasi utente di effettuare una richiesta di login alla piattaforma, tramite username e password;
- Consentire ad un utente, che ha effettuato l'accesso alla piattaforma, di depositare nel proprio conto di gioco un determinato importo maggiore di 0;
- Consentire a qualsiasi client di ricevere la lista degli eventi sportivi in corso;
- Consentire a qualsiasi client di ricevere un dettaglio di un particolare evento;
- Consentire a qualsiasi client di ricevere dettagli riguardo la cronologia degli eventi di un determinato Team;
- Consentire ad un utente, che ha effettuato l'accesso alla piattaforma, di aggiungere, modificare o rimuovere un evento ad una puntata di gioco in fase di creazione;
- Consentire ad un utente, che ha effettuato l'accesso alla piattaforma, di confermare la volontà di voler spendere del credito dal relativo conto di gioco per rendere effettiva la puntata di gioco;
- Consentire ad un utente, che ha effettuato l'accesso alla piattaforma, di ricevere un dettaglio riguardo la cronologia delle puntate effettuate;

In questo contesto sono state individuate le entità relative all'**utente**, **puntata di gioco** (e relativo **dettaglio**), **evento sportivo** (nel dominio affrontato sarà un incontro calcistico) e le relative squadre coinvolte.

2.2 Authentication Service

Questo servizio non è esposto direttamente ad un eventuale client ma viene utilizzato dell'`application-service` per sfruttarne le funzionalità esposte. Per garantire il corretto funzionamento della piattaforma di gioco, questo servizio deve implementare le seguenti funzionalità:

- Implementare le logiche necessarie alla funzionalità relativa ai meccanismi di registrazione e login;
- Implementare la logica necessaria alla funzionalità relativa al meccanismo di validazione di una richiesta ricevuta dall'`application-service`;
- Implementare la logica necessaria alla funzionalità relativa ai meccanismi di gestione degli utenti sulla piattaforma;
- Implementare le logiche necessarie alle funzionalità relative ai meccanismi di gestione dei conti di gioco di un utente sulla piattaforma;

In questo contesto sono state individuate le entità relative all'**utente** (informazioni di **registrazione**, **login** e **validazione**), **conto di gioco** e **transazione** (**prelievo** o **deposito** di credito).

2.3 Game Service

Questo servizio non è esposto direttamente ad un eventuale client ma viene utilizzato dall'`application-service` per sfruttarne le funzionalità esposte. Per garantire il corretto funzionamento della piattaforma di gioco, questo servizio deve implementare le seguenti funzionalità:

- Implementare la logica di ricezione delle informazioni relative agli eventi sportivi e alle relative squadre;
- Implementare le logiche che implementano le funzionalità necessarie a fornire le informazioni relative agli eventi (e squadre) all'`application-service`;

In questo contesto sono state individuate le entità relative agli **eventi sportivi** e **squadre** (teams).

2.4 Session Service

Questo servizio non è esposto direttamente ad un eventuale client ma viene utilizzato dell'`application-service` per sfruttarne le funzionalità esposte. Per garantire il corretto funzionamento della piattaforma di gioco, deve implementare le seguenti funzionalità:

- Implementazione della logica responsabile alla creazione o aggiornamento di una sessione utente;
- Implementazione della logica che fornisce all'`application-service` la sessione di un utente collegato alla piattaforma di gioco.

In questo contesto sono state individuate le entità relative ai **dati collegati alle attività di un utente** come il **dettaglio delle puntate di gioco**, in fase di creazione della scommessa.

3 Dettagli implementativi

3.1 Spring boot

Per quanto riguarda i servizi relativi al progetto, è stato scelto di utilizzare `Java 17` ed il framework `Spring Boot v.3.0.0`.

Ogni servizio implementa dei controller nel quale vengono definiti gli endpoint su cui vengono servite le funzionalità individuate dall'analisi dei requisiti funzionali.

La comunicazione tra questi servizi è ottenuta tramite l'utilizzo dell'implementazione di un *Client REST*, proprietario di Spring, `RestTemplate`. Ove necessario, tali richieste di comunicazione vengono manipolate da degli `Interceptor` customizzati, che permettono di aggiungere le informazioni necessarie con lo scopo, ad esempio, di validare la richiesta in modo tale che un microservizio possa accettare solo richieste da parte di determinato subset di ulteriori microservizi o client che conoscano la `Secret-Key`.

Le richieste effettate all'`application-service`, da parte dei client, sono validate tramite `BearerToken`, in questo scenario un utente che effettua l'accesso all'interno dell'applicativo viene verificato tramite `Token JWT`, implementato utilizzando un' apposita classe custom, che utilizza la libreria `io.jsonwebtoken`. Nello specifico, la maggior parte delle richieste in arrivo sono intercettate tramite l'utilizzo di classi che estendono l'implementazione `Filter` di Spring, `OncePerRequestFilter`, il comportamento di questi dipende dal caso di utilizzo e può essere diviso in due scenari:

- **Richiesta da un client all'**`application-service`: la richiesta è intercettata e il path richiesto è analizzato:

- Viene lanciato un errore se il path non è riconosciuto;

- Non viene effettuato alcun filtro se viene richiesta una API pubblica (individuate dal path `/app/public/*`), per il quale non è prevista alcuna limitazione di accesso;

- Viene effettuata una richiesta di validazione presso l'`authentication-service` contenente le informazioni necessarie (`Username` e `Authorization` estrapolati dagli header della richiesta), se questa da esito positivo la richiesta viene inoltrata al relativo controller, altrimenti non viene esaudita e una risposta di errore è ritornata al client;

- **Richiesta dall'**`application-service` **ad un microservizio interno**: viene effettuata la validazione della `Secret-Key` condivisa tra i due microservizi e, nel caso in cui questa dia esito positivo la richiesta è inoltrata al relativo *controller* altrimenti è ritornata una risposta di errore;

L'applicativo implementa anche dei sistemi di resilienza, fornite dal package `Resilience4j`. Nello specifico, sono implementati i design pattern **Circuit Breaker** e **Retry Pattern**, in combinazione tra loro. Questi permettono di evitare comportamenti errati nel caso di comunicazione instabile tra i microservizi e di evitare spreco di risorse interne e tempo di connessione ai client nel caso in cui parte dell'applicativo abbia subito un arresto anomalo. Le informazioni trasferite da e per ogni controller sono mappate tramite l'utilizzo di apposite classi denominiate `DTO` mentre le relazioni in database sono identificate tramite l'utilizzo della annotation `@Entity`, che permettono a Spring di mappare i record contenuti dei database all'interno delle relative classi. La generazione delle query ai vari database MySQL è demandata alla implementazione della interfaccia `JpaRepository`, un meccanismo per incorporare le funzionalità necessarie per la creazione, recupero e ricerca di dati, fornita dal package `JPA` di Spring. Vengono di conseguenza implementate delle classi `Converter` che si occupano della traduzione tra entità SQL e DTO e viceversa. Sono stati utilizzati dei package aggiuntivi, come `lombok`, che fornisce delle annotation utili a mantenere il codice leggibile e libero da codice boilerplate (eg. costruttori con parametri e senza, getter, setter e builder). In aggiunta è stata utilizzata la libreria `Swagger-UI` per la generazione automatica della documentazione riferita alle API di ciascun microservizio. Vengono utilizzate, inoltre, `JUnit` e `Mockito` per la creazione e gestione degli **Unit Test**.

3.2 Databases

Per la gestione dello storage è stato utilizzato `MySQL v 8.0`, come tecnologia per i database `SQL` e `Redis` per creare e gestire un database NoSQL contenente le informazioni di sessione dell'applicativo. Le tabelle e i relativi database prodotti per la realizzazione di questo progetto sono rappresentato dai seguenti diagrammi:

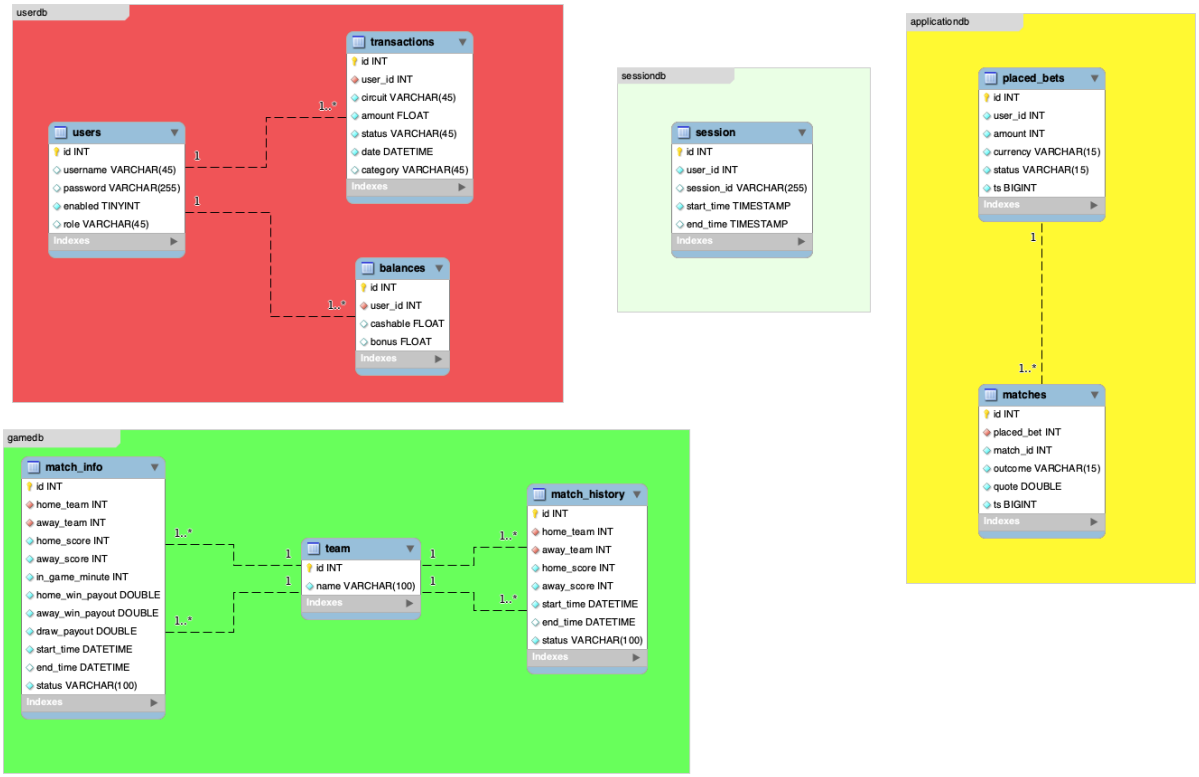

Figure 2: Schema ER dei databases

3.3 Docker

È stato utilizzato **Docker** per la creazione e gestione di container utilizzati da ogni servizio e database implementato. Ognuno di questi è a sua volta gestito dalla tecnologia **Docker Compose**, che ha permesso, tra gli altri dettagli, di definire eventuali variabili d'ambiente utilizzate nei servizi e delle configurazioni iniettate all'interno dei relativi container. In particolare, ogni container di servizio e storage implementato, può contenere all'interno del relativo file di configurazione `docker-compose.yml`, informazioni come:

- Immagine sorgente prelevata da *Docker Hub*;
- Volumi di persistenza dei dati;
- Variabili d'ambiente come indirizzi di riferimento per la comunicazione con i database e relative credenziali;
- Reti collegate ai relativi servizi, ognuno dei quali appartiene al numero minore di reti necessarie a garantirne il funzionamento, in questo modo si preclude l'accesso ad un container verso altri componenti su cui non è predisposta dall'applicazione alcuna interazione;
- Porte esposte sull'host del servizio docker, utilizzate per accesso remoto e debug;

3.4 Git e Continuous Integration

Per la gestione del *versioning* del codice prodotto è stata utilizzato *git*, i relativi *repository* sono ospitati sulla piattaforma **GitHub**. Ogni microservizio e database prodotto dispone di un proprio repository. Per ognuno di questi è stato implementato un **workflow per la CI**, questo si occupa di creare le immagini definire dai `Dockerfile` presenti in ogni progetto che verranno caricate dallo stack di container implementato tramite `Docker Compose`.

Prima della pubblicazione dell'immagine (su Docker Hub), è necessario che il codice implementato passi degli **unit test** per verificare il corretto funzionamento di parti di programma permettendo così una precoce individuazione dei bug e degli errori prima del deploy.

3.5 Esempio flusso di `/app/gamble/add`

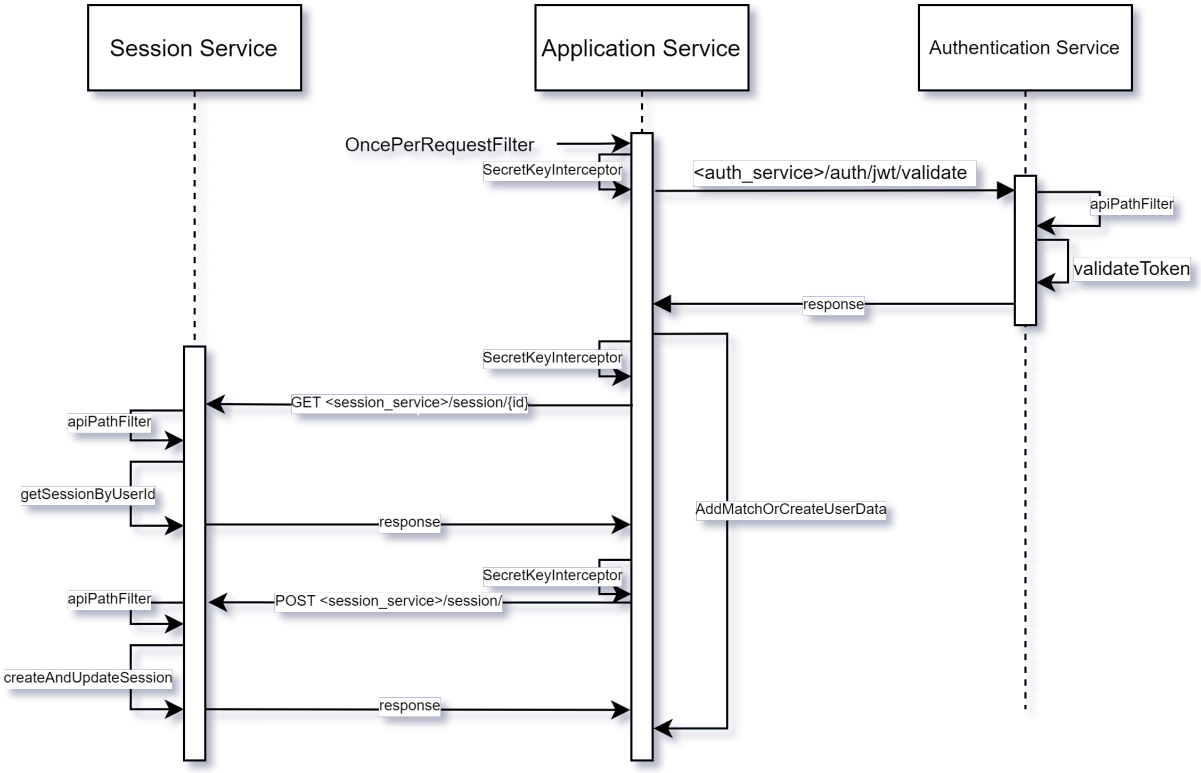

Figure 3: Diagramma rappresentate il flusso di chiamate scatenate dall'invocazione dell'**API** `/app/gamble/add` da parte di un utente collegato alla piattaforma.

L'esempio mostrato è utile per rappresentare la logica che sta dietro la gestione della sessione e dell'autenticazione. Si suppone che un utente, dopo essersi loggato, aggiunga ad una delle proprie schedine un nuovo evento o ne crei una da zero. Il flusso può essere descritto come segue:

- La richiesta effettuata dal client viene intercettata dal filtro che estende `OncePerRequestFilter`, questa si occupa di effettuare **POST** all'`authentication-service` con le informazioni relative a `Username` e `Bearer token` (sull'Header). Quest'ultima è intercettata dal `SecretKeyInterceptor` che si occupa di iniettare al suo interno le informazioni riguardanti la `Secret-Key` in modo da validare la comunicazione tra `application-service` e l'`authentication-service`;
- L'`authentication-service` invoca il metodo `validateToken` che effettua i controlli sui dati ricevuti, ne valida la correttezza e invia una risposta all'`application-service`. Quest'ultimo, se la verifica è andata a buon fine, concede alla request di essere consumata dal relativo controller;
- Il controller che ha il compito di rispondere alle richieste per il path `/add/gamble/add` invoca il metodo `addMatchOrCreateUserData` che a sua volta effettua le seguenti operazioni:

- Effettua richiesta GET al `session-service`, anche questa è intercettata dal `SecretKeyInterceptor`;
- La richiesta è intercettata, nel `session-service` dall' `apiPathFilter` che si occupa della sua validazione tramite `Secret-Key`, in caso di esito positivo, il relativo controller viene autorizzato a consumare la richiesta;
- Il controller sul `session-service` invoca il metodo `getSessionByUserId` che si occupa di instaurare una connessione prima con il database `sessiondb`, per verificare se è presente una sessione aperta relativa all'utente e, con le informazioni ricevute, con il servizio **Redis**, per recuperarla o crearla. Questa viene ritornata al controller e di conseguenza è inoltrata all'`application-service`;
- Il metodo `addMatchOrCreateUserData` continua con l'elaborazione dei dati ricevuti dal `session-service`, aggiungendo o modificando le informazioni contenute in sessione secondo la richiesta ricevuta dal client;

- Una volta ultimate, queste modifiche vengono reinoltrate al `session-service` tramite l'invocazione di una nuova richiesta POST (anche quest'ultima è intercettata da entrambi i servizi dai relativi filtri);

- Il `session-service` in risposta a tale richiesta invoca il metodo createAndUpdateSession che si occupa dell'aggiornamento del database sessiondb e della sessione contenuta sullo storage Redis;

- Il risultato di tali aggiornamenti verrà ritornato come risposta al client che ha inoltrato la richiesta iniziale;