[Introduzione 1](#_Toc107662299)

[Analisi dei requisiti e progettazione 2](#_Toc1554951829)

[Architettura 2](#_Toc934531207)

[Dati 3](#_Toc289999740)

[Servizio BE Postgres 4](#_Toc939131923)

[Soluzione 4](#_Toc1400188944)

[Struttura progetto 4](#_Toc196445128)

[Difficoltà 5](#_Toc1918863550)

[Requisiti 5](#_Toc2036618094)

[Limitazioni 5](#_Toc1775189388)

[Servizio BE MainServer 5](#_Toc866180890)

[Soluzione 5](#_Toc708879321)

[Difficoltà 6](#_Toc819845946)

[Requisiti 6](#_Toc1367190558)

[Limitazioni 6](#_Toc774496948)

[Struttura progetto 6](#_Toc1997461613)

[Dashboard 7](#_Toc479297005)

[Soluzione 7](#_Toc973651631)

[Difficoltà 7](#_Toc1132353541)

[Requisiti 7](#_Toc1073596333)

[Limitazioni 7](#_Toc326930462)

[Analisi dati (jupyter notebook) 7](#_Toc905424129)

[Soluzione 8](#_Toc1250959410)

[Difficoltà 8](#_Toc644027667)

[Requisiti 8](#_Toc932595528)

[Limitazioni 8](#_Toc1749933772)

[Conclusioni 8](#_Toc1668914538)

[Divisione del lavoro 8](#_Toc1409248788)

[Informazioni extra 8](#_Toc149441253)

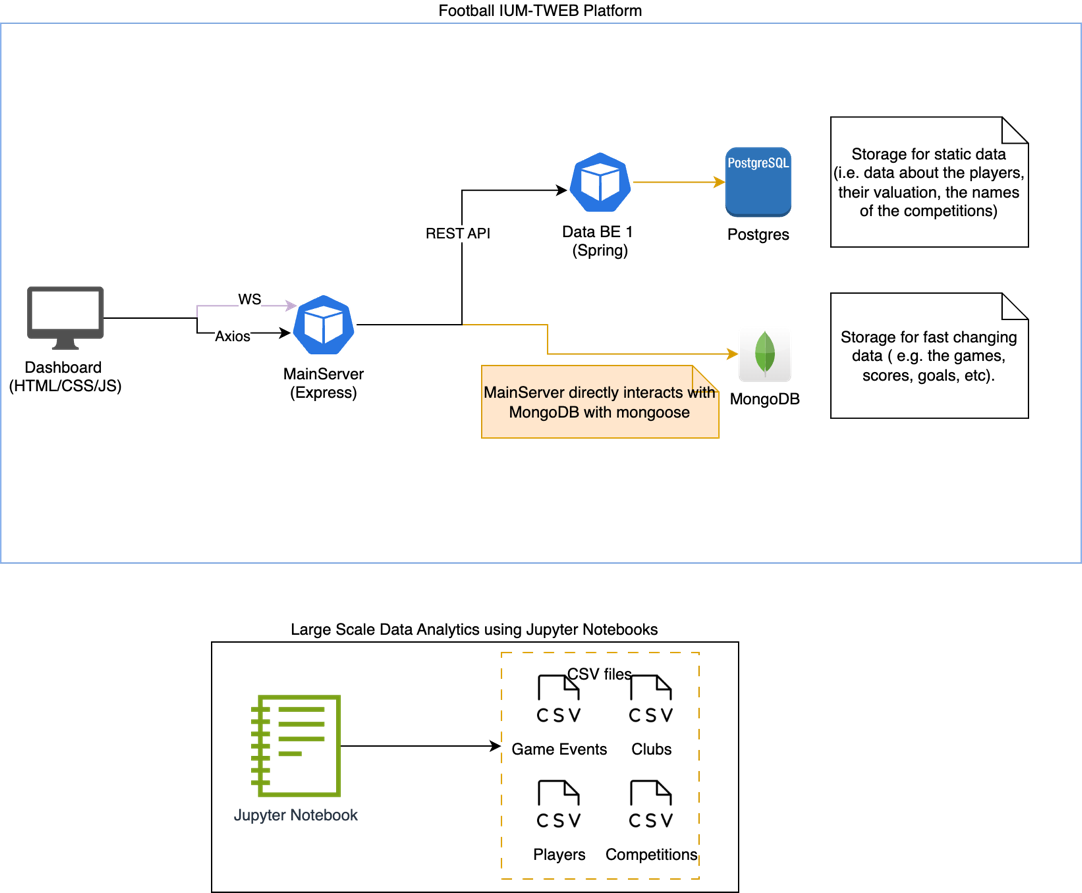
[Bibliografia 9](#_Toc1640911533)

# Introduzione

Il report contiene informazioni relative al design, scelte architetturali, strutturazione del progetto, limitazioni della piattaforma e ostacoli affrontati durante il processo.

# Analisi dei requisiti e progettazione

## Architettura



Come mostrato in figura, l’architettura è composta da 5 attori principali:

1. Un database relazionale, postgres, che contiene i dati relazionali a basso tasso di cambiamento
2. Un database non-relazionale documentale, mongoDB, usato per immagazzinare dati che cambiano frequentemente
3. Un servizio, scritto in java e che fa uso del framework SpringBoot, che fa da interfaccia verso il database relazionale. Tutte le richieste che avranno necessità di accedere a tali dati transiteranno da questo servizio.
4. Un applicativo, scritto in javascript e che utilizza express e nodeJs, che si occupa di:
   1. fare il routing verso l’altro servizio per le richieste che necessitano di accedere a dati immagazzinati su postgres
   2. Eseguire interrogazioni al database mongoDB
   3. Gestire le connessioni websocket per la chat
5. Il client, che consuma tali API tramite una dashboard sviluppata in html/js/css e si connette al server socket.io per la comunicazione real-time

## Dati

Una parte rilevante della progettazione è stata quella legata alla suddivisione del dataset tra Postgres e MongoDB:

* competitions.csv  
  DB: Postgres  
  Motivazione: Le modifiche avvengono molto raramente
* game\_lineups.csv  
  DB: Postgres  
  Motivazione: Cambia quasi mai, le righe vengono inserite quasi sempre una sola volta (prima della partita)
* player\_valuations.csv  
  DB: Postgres  
  Motivazione: Cambia quasi mai, le righe vengono inserite quasi sempre una sola volta (dopo la partita)
* clubs.csv  
  DB: Postgres  
  Motivazione: Necessita di aggiornamenti rari poiché contiene dati statici sui club
* players.csv  
  DB: Postgres  
  Motivazione: Contiene dati statici sui giocatori
* club\_games.csv  
  DB: Postgres  
  Motivazione: Aggiornata solo a fine partita
* appearances.csv  
  DB: MongoDB  
  Motivazione: I suoi campi vengono spesso aggiornati (goal, cartellini, minuti giocati, ...)
* game\_events.csv  
  DB: MongoDB  
  Motivazione: Contiene tutti gli eventi che accadono in una determinata partita, uno dei più frequenti
* games.csv  
  DB: MongoDB  
  Motivazione: I suoi campi vengono spesso aggiornati e c'è anche ridondanza (ad esempio, nomi dei club)

Un ulteriore compito svolto è stato quello legato alla creazione degli schema su Postgres verificando il tipo corretto per ogni colonna.

# Servizio BE Postgres

## Soluzione

La soluzione proposta consiste in un servizio java, che fa uso di springboot come framework. Il codice è stato sviluppato così da essere quanto più ordinato, modulare, estendibile e ben documentato.

Particolare attenzione è stata rivolta all’organizzazione del codice, che è riportata più in basso. Alcune considerazioni chiave:

* Per il livello di servizio è stata definita un’interfaccia per ogni funzionalità (es. ClubService). Ed è stata fornita una implementazione (ClubServiceImpl). In tal modo si può disacoppiare l’interfaccia, ovvero il patto che si ha con il consumatore di tale servizio, e la sua implementazione può quindi cambiare nel tempo senza ripercussioni. I controller hanno anche il supporto al versioning delle API.
* Creazione di eccezioni custom e la gestione delle eccezioni.
* Codici di risposta HTTP consoni (es. Se una risorsa di una chiamata GET ad un endpoint non è esistente, 404. Per errori server 500, etc).
* Dependency injection, tramite autowired e constructor injection, per la gestione delle dipendenze tra i vari servizi, controller, etc.
* Definizione di modelli di dati e utilizzo di chiavi composite per le entità che non avevano un unico campo chiave
* Creazione di DTO per risposte a chiamate che non combaciano con il modello di dati soggiacente.
* Configurabilità: Mediante application.properties o variabili di ambiente è possibili modificare vari parametri come quelli di connessione al db, gli endpoint swagger, la porta di ascolto del server.
* La documentazione è presente sia in forma di docstring per tutto il livello di servizio, sia mediante swagger-ui grazie alla dipendenza springdoc-openapi-starter-webmvc-ui. Runnando in locale, è possibile reperire la swagger-ui a [questo indirizzo](http://localhost:8989/swagger-ui.html) e il file json openAPI 3.0 a [quest’altro](http://localhost:8989/api-docs).

### Struttura progetto

* Controller
* Data
  + Model
  + Output
* Exception
* Repository
* Service
  + Impl

## Difficoltà

Il task richiedeva la creazione di un applicativo che fosse in grado di inviare interrogazioni al database relazionale e offrire delle API per tale scopo. Sono state affrontate alcune sfide, tra cui:

* La necessità di definire chiavi composite in quanto alcune tabelle non avevano un unico campo candidato a chiave (e quindi univoco). Un esempio è dato da *PlayerValuation* in quanto per un dato playerId si può avere più di un match. Il problema è stato affrontato con l’uso dell’annotazione EmbeddedId che consente di definire chiavi composte da più campi.
* Gestione di campi opzionali nei controller necessario per i filtri (ricerca per data, paginazione)
* Query complesse che richiedevano join tra più tabelle (es. *PlayerValuationRepository.findTopValuationsWithPlayerInfoByDates*)
* Paginazione dei risultati

## Requisiti

I requisiti sono stati soddisfatti sia in termini tecnologici (linguaggio, framework e dipendenze utilizzate) sia in termini di qualità del codice e documentazione.

## Limitazioni

Una delle maggiori limitazioni del sistema è data dalla mancanza di autenticazione, gestione di ruoli o permessi. Attualmente tutte le API possono essere chiamate da tutti.

Assenza possibilità per modificare oggetti parzialmente (es. Metodo REST PATCH)

# Servizio BE MainServer

## Soluzione

Il servizio è stato sviluppato in nodeJS mediante il framework express.

Particolare attenzione è stata rivolta all’organizzazione del codice, che è riportata più in basso. Alcune considerazioni chiave:

* Utilizzo di codice asincrono e non bloccante mediante promesse consumate con await e then. Per il forward delle richieste si è utilizzato fetch.
* La documentazione è presente sia in forma di commenti e principalmente mediante swagger-ui grazie alle dipendenze swagger-jsdoc e swagger-ui-express. Runnando in locale, è possibile reperire la swagger-ui a [questo indirizzo](http://localhost:3000/api-docs/) e il file json openAPI 3.0 a [quest’altro](http://localhost:3000/swagger.json).
* Codici di risposta HTTP conforme allo standard REST(es. Se una risorsa di una chiamata GET ad un endpoint non è esistente, 404. Per errori server 500, etc).
* Utilizzo di query complesse per aumentare l’efficienza del codice e recuperare dati di più collezioni contemporaneamente.
* Creazione di DTO custom per risposte a chiamate che non combaciano con il modello di dati soggiacente.
* Configurabilità: Mediante config.js o variabili di ambiente: è possibile modificare vari parametri come quelli di connessione al db, il livello di verbosità dei log, la porta di ascolto del server.

## Difficoltà

Il task richiedeva la creazione di un applicativo che fosse in grado di svolgere varie funzioni: fare da proxy per inoltrare tutte le richieste relative a dati relazionali all’apposito servizio, offrire una interfaccia per eseguire interrogazioni su mongoDB, eseguire socket.io per comunicazione in tempo reale (es. Chat).

Sono state affrontate alcune sfide, tra cui:

* L’utilizzo di aggregate per eseguire query a mongoDB complesse che prevedevano il reperimento di dati tra più tabelle e ottenimento di risultati aggregati.
* Paginazione
* Gestione di campi opzionali nei controller necessari per i filtri (ricerca per data, paginazione)
* Gestione codici di risposta
* Configurazione swagger e politiche CORS

## Requisiti

I requisiti sono stati soddisfatti sia in termini tecnologici (linguaggio, framework e dipendenze utilizzati) sia in termini di qualità del codice e documentazione.

## Limitazioni

Anche in questo caso manca del tutto un meccanismo di autenticazione e controllo degli accessi. Inoltre, i messaggi della chat, anche se supportano diverse stanze, sono filtrati a FE. Questo significa che i messaggi possono potenzialmente essere letti da partecipanti non nella stessa stanza.

Assenza di possibilità di modificare oggetti parzialmente (es. Metodo REST PATCH)

### Struttura progetto

* Config
* Controllers
* Databases
* Models
* Routes
* Services

# Dashboard

## Soluzione

Per rispettare i requisiti tecnologici è stata sviluppata una dashboard facendo uso delle seguenti tecnologie: html/css/js. Inoltre si è fatto uso di bootstrap per la parte grafica, axios per le richieste HTTP e socket.io per la gestione delle websocket (chat).

## Difficoltà

L'obiettivo del task era fornire ai tifosi e agli esperti la possibilità di accedere a dati sul calcio. È stato quindi progettata una dashboard con diverse pagine, contenenti tabelle e grafici, nonchè sistemi di filtraggio dati. Inoltre, uno dei requisiti era quello di avere una chat con supporto a diverse stanze.

Sono state affrontate alcune sfide, tra cui:

* L’ottimizzazione delle prestazioni
* L’implementazione della chat e relative grafiche e azioni (animazione e contatore nuove messaggio, suono di notifica)
* Filtraggio in real time man mano che l’utente scrive nei campi di ricerca nella pagina di PlayerStats
* Lo sviluppo dell’interfaccia responsive mediante grid e flex layout ed elementi “card”
* Modifica dei componenti grafici dinamicamente

## Requisiti

I requisiti sono stati soddisfatti sia in termini tecnologici (linguaggio, framework e dipendenze utilizzati) sia in termini di qualità del codice.

## Limitazioni

La pagina di registrazione non funziona ed è stata inclusa per la parte IUM.

Due utenti possono scegliere lo stesso username in chat, causando confusione.

Bisogna inserire manualmente il nome della stanza in cui l’utente vuole entrare.

# Analisi dati (jupyter notebook)

## Soluzione

Si è utilizzato un jupyter notebook e in particolare le librerie pandas e seaborn.

L’analisi dei dati è stata effettuata sui diversi file csv del dataset. Si è inoltre cercato di rimanere consistenti con gli altri task, ad esempio è stata creata una formula ad-hoc per il reperimento di giocatori “sottovalutati” considerando il loro valore di mercato e il punteggio del fantacalcio che avrebbero generato in una determinata partita.

## Difficoltà

L'obiettivo del task era quello di esplorare il dataset ed estrarre informazioni.

Sono state affrontate alcune sfide, tra cui:

* Gestione dei tipi
* Uso di LabelEncoder per alcuni campi stringa (es. Ruolo del giocatore)
* Composizione di query più complesse
* Composizione di grafici con dati provenienti da più tabelle
* Utilizzo di chunksize e altre tecniche di ottimizzazione (es. Drop di colonne non richieste) per join particolarmente onerosi che altrimenti generavano errore di out of memory.

## Requisiti

I requisiti sono stati soddisfatti sia in termini tecnologici (linguaggio e dipendenze utilizzate) sia in termini di risultati ottenuti.

## Limitazioni

Data cleaning superficiale.

L’analisi dei dati non è esaustiva.

Alcuni grafici sono espressi in termini assoluti e non relativi (senza normalizzazioni).

# Conclusioni

Una delle sfide più grandi è stata sicuramente la parte di progettazione, architettura e sviluppo modulare del codice. Un’altra parte che ha richieste notevole tempo è legata alle prestazioni e all’utilizzo di meccanismi asincroni.

Sono state inoltre affrontate varie difficoltà come evidenziato nei capitoli legati ai singoli task.

# Divisione del lavoro

N/A essendo il gruppo costituito da un solo membro.

# Informazioni extra

Per utilizzare la dashboard è necessario avviare i due servizi BE. Inoltre, per fare ciò risulta necessario configurare opportunamente i dati di connessione al DB ed avere i database online.

Poichè solo il di postgres è fornito (dump intero e solo schema), qualora si volesse replicare l’ambiente mongo è possibile farlo eseguendo i seguenti comandi:

* mongoimport --uri <mongoDBURI> --collection games --type csv --headerline --file games.csv --ignoreBlanks
* mongoimport –uri <mongoDBURI> --collection game\_events --type csv --headerline --file game\_events.csv --ignoreBlanks
* mongoimport --uri <mongoDBURI> --collection appearances --type csv --headerline --file appearances.csv --ignoreBlanks

Il codice sviluppato è contenuto in tre repo github :

* <https://github.com/clark1492/IUM_TWEB_FE>
* <https://github.com/clark1492/IUM_TWEB_BE_DataPos>
* <https://github.com/clark1492/IUM_TWEB_BE_MainServer>
* <https://github.com/clark1492/IUM_TWEB_Notebook>

# Bibliografia

* <https://www.baeldung.com/jpa-composite-primary-keys>
* <https://www.baeldung.com/spring-data-jpa-pagination-sorting>
* <https://github.com/axios/axios#example>
* <https://www.geeksforgeeks.org/spring-dependency-injection-autowired-vs-constructor-injection/>
* <https://dev.to/kabartolo/how-to-document-an-express-api-with-swagger-ui-and-jsdoc-50do>
* <https://socket.io/docs/v4/>
* <https://mongoosejs.com/docs/api/aggregate.html>
* <https://stackoverflow.com/questions/75577333/swagger-json-file-download-on-nodejs-api>
* <https://www.w3schools.com/jsref/met_audio_play.asp>