

Sistemi distribuiti

Maraffa Online

Sofia Tosi Matteo Santoro

Supervisors: Giovanni Ciatto, Matteo Magnani, Andrea Omicini

13 gennaio 2025

Indice

0.1	List of Abbreviations	iii
0.2	Glossario dei termini	iii
1	Introduzione	1
2	Analisi	2
2.1	Obiettivi	2
2.2	Politiche di autovalutazione	2
2.3	Requisiti e casi d'uso	3
2.3.1	Requisiti	3
2.3.2	Casi d'uso	3
3	Architettura e Design	5
3.1	Design generale del software	5
3.2	Micro-servizi e la loro architettura	6
3.2.1	Servizio per Gestione Utenti	7
3.2.2	Servizio Business Logic	7
3.2.3	Servizio ponte tra i microservizi	7
3.3	Perche' i microservizi?	7
3.3.1	Vantaggi dell'architettura a microservizi	8
3.3.2	Struttura API Gateway	9
3.3.3	WebSockets	10
4	Implementazione	11
4.1	Componenti	11
4.1.1	Business Logic	11
4.1.2	User Service	11
4.1.3	Front-End	11
4.1.4	Middleware	12
4.2	Tecnologie	13
4.2.1	Swagger - OpenAPI	13
4.2.2	Postman	14
4.2.3	Log automatici	15
4.2.4	Checkstyle, javadoc, github page	16
4.2.5	Lettura variabili d'ambiente	18
4.3	Riflessioni sulla politiche di autovalutazione	18
4.4	Mockup	19
4.5	Screen	19

5	Interazioni	28
5.1	Frontend ↔ WebSocket	28
5.1.1	Chat globale e privata	28
5.1.2	Reattività e aggiornamento in tempo reale	28
5.2	Stack Docker → reti interne e IP privati	29
5.2.1	Nginx	29
6	DevOps	30
6.1	DVCS	30
6.1.1	Struttura dei Branch	30
6.1.2	Conventional commits	30
6.2	Issue template	31
6.3	Build automation	31
6.3.1	Scheletro del workflow	31
6.3.2	UserManagement e BusinessLogic	32
6.3.3	Middleware	32
6.3.4	Frontend	32
6.4	Continuous integration	32
7	Validazione	33
7.1	Test	33
7.1.1	Testing asincrono	34
7.2	Code quality e linting	35
8	Deployment	37
8.1	Containerization	37
8.1.1	NodeJS	38
8.1.2	Java	38
8.1.3	Angular	39
8.2	Monitoraggio	39
8.2.1	Grafana	39
8.2.2	Prometheus	39
8.2.3	cAdvisor	40
8.2.4	Integrazione e Configurazione	40
8.2.5	Esecuzione del sistema	41
9	Conclusions and Future Work	42
9.1	Conclusioni	42

Elenco delle figure

2.1	Casi d'uso	4
4.1	Mockup della schermata di Login e Registrazione	19
4.2	Mockup della dashboard	19
4.3	Mockup cambio password	20
4.4	Mockup della Waiting Room	20
4.5	Mockup del profilo dell'utente	21
4.6	Mockup della schermata di gioco	22
4.7	Screen della schermata di Login e Registrazione	23
4.8	Screen della schermata Registrazione	24
4.9	Screen Dashboard	25
4.10	Screen cambio password	25
4.11	Screen della schermata waiting room	26
4.12	Screen del profilo dell'utente	26
4.13	Screen della schermata di gioco	27
8.1	Comparazione tra le immagini multi-stage e le immagini tradizionali	37

Elenco delle tabelle

3.1	Tabella descrittiva User Service	7
3.2	Tabella descrittiva Business Logic	7
3.3	Tabella descrittiva Middleware	7

0.1 List of Abbreviations

SMPCS	School of Mathematical, Physical and Computational Sciences
CS	Computer Science
ghP	GitHub Pages
CI	Continuous Integration

0.2 Glossario dei termini

Nome	Descrizione	Sinonimi
Mano	Distribuzione delle 40 carte ai 4 giocatori e la seguente serie di 10 prese	Round
Mano	Carte dei giocatori non ancora giocate	Hand
Preso	Quando ogni giocatore, a turno, gioca sul tavolo una carta. L'ultima presa della mano vale 1 punto.	Trick
Partita	Insieme di più mani fino al raggiungimento del punteggio di 41 punti.	Game
Partita corta	Insieme di più mani fino al raggiungimento del punteggio di 31 punti.	Short Game
Tavolo	Raggruppamento di 4 giocatori, suddivisi in 2 coppie, i giocatori della stessa squadra "siedono" in direzione opposta	Table
Seme	Tipologia distintiva di carta, ne esistono 4: Denari, Coppe, Spade, Bastoni	Suit: Coins, Cups, Swords, Clubs
Briscola	Seme con priorità più alta.	Trump
Maraffa	Se un giocatore possiede le tre carte di valore maggiore (asso, due e tre, dette assieme "Maraffa" o "Cricca") del seme di briscola, vince tre punti aggiuntivi. In questo caso deve scendere con l'asso di quel seme.	Cricca, Marafon, Tresette con la Briscola
Mazzo	40 carte, di 4 semi diversi, 1,2,3,4,5,6,7, fante, cavallo e re.	Deck
Taglio	Durante una mano in un seme viene giocato il seme di briscola, che avendo priorità maggiore permette di prendere nonostante il seme di gioco	Cut
Busso	Invita il compagno, se possibile, a conquistare la presa e ad aprire il turno successivo con lo stesso seme	Knock
Striscio corto	Quando si ha ancora in mano un basso numero di carte dello stesso seme con cui si è aperto il turno.	Short strip
Striscio lungo	Quando si ha ancora in mano molte carte dello stesso seme con cui si è aperto il turno.	Long strip
Volo	Quando non si hanno più carte del seme con cui si è aperto il turno.	Fly
Figura	Fante, Cavallo, Re, con punteggio di 1/3 di punto.	Figure
Asso	Carta con valore di 1 punto.	Ace
Due e Tre	Carte con valore 1/3 di punto.	Two and Three
Carta liscia	Carte con numeri 4, 5, 6, 7. Sono prive di valore	Smooth paper
Squadra	Coppie di giocatori seduti opposti	Team
Giocatore	Persona che interagisce con l'applicativo	Player, User
Chiamata fuori	Se un giocatore pensa che la sua squadra abbia raggiunto i 41 punti (o 31 punti nella variante "corta" della partita), la squadra può dichiarare di avere già nel mazzo delle prese i punti per vincere e chiudere in anticipo l'ultima partita. In questo modo la mano termina immediatamente, senza che vengano giocate le restanti prese e la squadra che si è "chiamata fuori" impedisce all'altra squadra di conquistare ulteriori prese. Se una squadra si chiama fuori e, dopo aver contato i punti delle prese effettuate ed averli sommati ai punti ottenuti nelle mani già giocate, non raggiunge i punti per la vittoria (in gergo "sbaglia la chiamata") scatta automatico l'11 a 0 per la squadra avversaria	Call out
Modalità di gioco	regole di gioco classiche o varianti che influenzano aspetti come il punteggio, condizioni di vittoria/perdita, ... Ne sono state implementate due: classica, 11 a 0.	Game mode

Capitolo 1

Introduzione

MaraffaOnline è un'applicazione che permette alle persone di giocare al gioco di carte Maraffa/Beccacino. Il progetto consiste nell'eseguire una manutenzione evolutiva del gioco di carte reperibile su MaraffaOnline, attualmente sviluppato dalla prof.ssa Lumini. In particolare la nuova versione avrà un'architettura a microservizi e introdurrà anche nuove funzionalità come formazione personalizzata delle squadre, una nuova modalità di gioco (vittoria 11 a 0 in caso di violazione delle regole da parte di una squadra), salvataggio delle statistiche delle partite e degli utenti, ...

Per realizzare questa evoluzione, il progetto sarà suddiviso in diverse fasi, ognuna delle quali si concentrerà su un aspetto specifico dell'architettura e delle funzionalità.

La prima fase riguarderà la decomposizione dell'applicazione monolitica esistente in microservizi indipendenti, che comunicheranno tra loro attraverso API RESTful.

Nella seconda fase, saranno sviluppati i sistemi di salvataggio delle statistiche e la chat. Le statistiche di gioco saranno memorizzate in un database relazionale, permettendo agli utenti di visualizzare le loro performance nel tempo e confrontarsi con altri giocatori. Il gioco disporrà di due chat: una globale, accessibile a tutti gli utenti, e una relativa alla partita, accessibile solo ai giocatori della partita in corso.

La terza fase si concentrerà sull'implementazione delle nuove funzionalità, partendo dalla formazione personalizzata delle squadre fino alla modalità 11 a 0, la quale assicurerà che le violazioni delle regole siano rilevate automaticamente.

Il progetto sarà gestito utilizzando metodologie agili, con sprint di sviluppo e feedback continuo da parte degli utenti. In questo modo, sarà possibile adattare rapidamente il piano di lavoro in base ai consigli, garantendo al contempo un prodotto finale di alta qualità e in linea con le aspettative degli utenti.

Capitolo 2

Analisi

2.1 Obiettivi

I principali requisiti funzionali e obiettivi del sistema sono:

- **Costruzione di un'architettura a micro-servizi.**
- **Autenticazione e registrazione utenti:** possibilità di accedere ad un'area privata per poter visualizzare informazioni personali, oltre alla possibilità di poter giocare senza un proprio account in modalità 'ospite'.
- **Creazione e gestione partita:** gli utenti possono creare o partecipare ad una partita che inizierà soltanto quando 4 giocatori saranno presenti e disposti in due squadre da 2.
- **Chat di gioco:** possibilità di scambiare messaggi.
- **Nuova modalità di gioco** in cui se si commette un errore di gioco la partita viene conclusa conferendo alla squadra avversaria il massimo dei punti.
- **Persistenza dei dati:** salvataggio su database degli utenti e delle statistiche delle partite (mani giocate, briscola, etc...).

2.2 Politiche di autovalutazione

Di seguito vengono riportati i casi che dovranno essere gestiti dal sistema:

- l'elenco delle partite iniziate/finite deve rimanere aggiornato
- se un giocatore si disconnette la partita termina per tutti
- connessione lenta o malfunzionamenti da parte di un giocatore
- un giocatore clicca sul tasto "esci"
- nel caso di un crash di un servizio si deve garantire che il servizio venga riavviato in automatico evitando che il sistema si blocchi

2.3 Requisiti e casi d'uso

2.3.1 Requisiti

1. Account
 - (a) Login
 - (b) Registrazione
 - (c) Recupero password
 - (d) Visualizzazione profilo
 - (e) Modifica password
 - (f) Possibilità di scegliere se giocare come ospite o effettuare il login
2. Realizzazione partita
 - (a) Creazione partita
 - (b) Partecipazione partita
 - (c) gioca carta
 - (d) Inizio partita
 - (e) Fine mano
 - (f) Fine partita
3. Chat di gioco
 - (a) chat globale
 - (b) chat partita
4. Possibilità di scegliere un compagno di squadra
5. Scelta del seme, parole consentite
6. Modalità di gioco 11 a 0
7. Gestione punteggio
 - (a) Calcolo totale e parziale (Gestione per ogni mano) del punteggio
 - (b) Maraffa/Cricca (+3 punti)
8. Servizio gestione utenti
9. Salvataggio statistiche
10. Realizzazione GUI
 - (a) Refactor della GUI esistente
 - (b) Rinnovamento GUI

2.3.2 Casi d'uso

Si riporta di seguito lo schema dei casi d'uso che modella l'interazione dell'utente con l'applicazione.

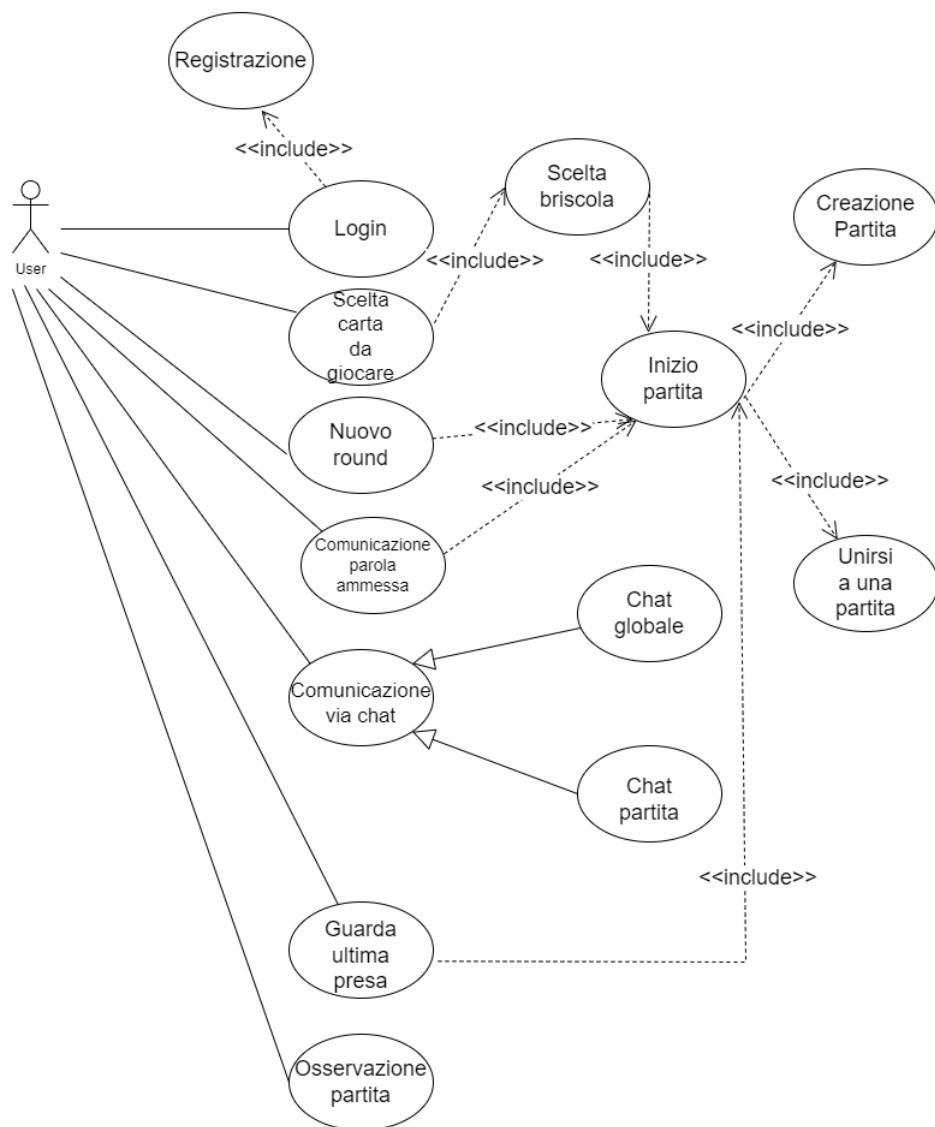


Figure 2.1: Casi d'uso

Capitolo 3

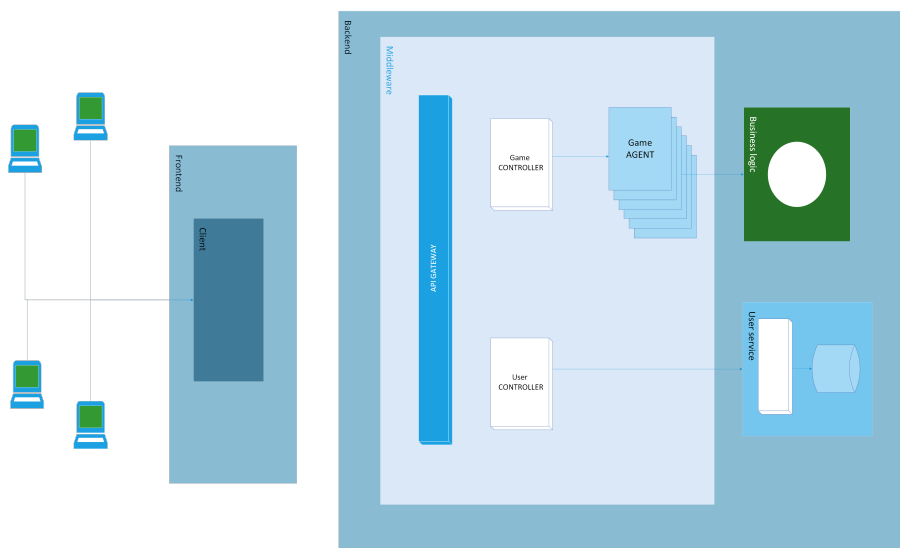
Architettura e Design

3.1 Design generale del software

L'architettura del software è basata su micro-servizi autonomi e indipendenti tra loro e la loro comunicazione avviene tramite API REST. Esiste un middleware che si occupa della gestione e dell'orchestrazione dei micro-servizi, facendo appunto da ponte collegando le varie diverse tecnologie.

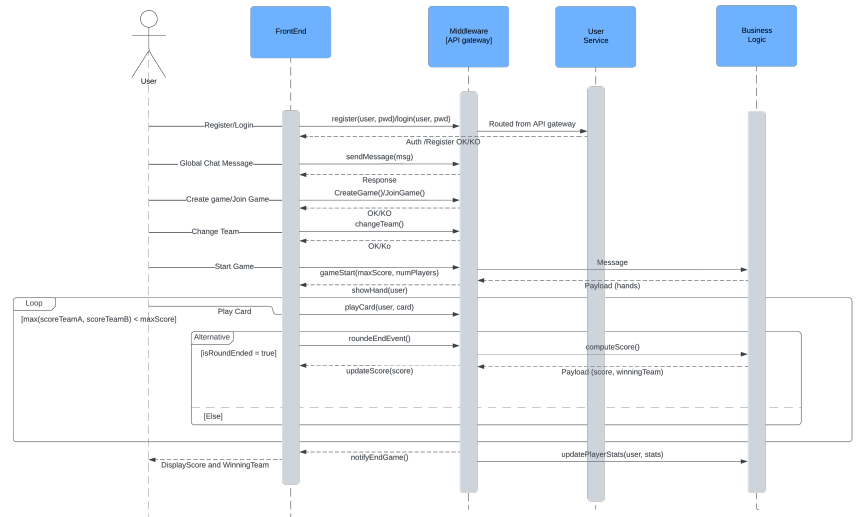
I microservizi che si sono scelti di sviluppare si occupano di varie aree legate alla natura del progetto e sono:

- **UserService:** si occupa della gestione degli utenti, quindi la loro registrazione, autenticazione e gestione dei dati personali.
- **BusinessLogic:** si occupa di mantenere al proprio interno tutte le regole proprie del gioco
- **Front-End:** si occupa di gestire l'interfaccia grafica e la comunicazione con il middleware.
- **Middleware:** si occupa della gestione delle comunicazioni tra i vari microservizi, e svolge la funzione di "motore di gioco".



Le interazioni tra i microservizi avvengono prevalentemente tramite API rest e websocket. Qui di seguito uno sequence diagram che riassume come avviene lo scambio di informazioni tra i vari componenti.

All'interno del capitolo dedicato alle Iterazioni quest'ultime verranno approfondite in dettaglio.



C03E12

3.2 Micro-servizi e la loro architettura

3.2.1 Servizio per Gestione Utenti

Table 3.1: Tabella descrittiva User Service

Linguaggio	Node.JS/TypeScript
Framework	NestJS
Persistence	Mysql
Librerie	TypeORM
Architettura	Model-Controller-service

3.2.2 Servizio Business Logic

Table 3.2: Tabella descrittiva Business Logic

Linguaggio	Node.JS/TypeScript
Framework	NestJS
Architettura	Model-Controller-service

3.2.3 Servizio ponte tra i microservizi

Table 3.3: Tabella descrittiva Middleware

Linguaggio	Java
Server	Vertx
Librerie	RabbitMQ(vertex)
Architettura	Multi-agent, motore di gioco

3.3 Perché i microservizi?

Ogni microservizio si occupa di gestire un singolo ambito di progetto, il che fa sì che si ottengano ottimi risultati riguardo la qualità e l'affidabilità, la riutilizzabilità, la flessibilità e la scalabilità. Tutti i componenti sono "loosely coupled", distribuiti in modo indipendente, altamente collaudabili e gestibili e in ambiti reali gestiti da un piccolo team. Tuttavia, non si tratta di una soluzione definitiva, ha anche degli svantaggi come ad esempio:

- la complessità di gestione e la necessità di un'infrastruttura di rete molto complessa.
- un'ottima comunicazione di team.
- una discreta capacità di astrazione e progettazione di interfacce di comunicazione che devono essere rispettate da tutti i microservizi.
- Non si ha la certezza che l'infrastruttura funzioni correttamente fino a quando tutti i servizi non cooperano tra loro.

3.3.1 Vantaggi dell'architettura a microservizi

Fault Tolerance

Progettare un sistema completamente privo di errori è un obiettivo decisamente ambizioso e molto difficile da raggiungere. Molto più realistico, e inerente al nostro percorso, è progettare un sistema che sia in grado di tollerare, nella giusta misura, gli errori che inevitabilmente si verificheranno. La scelta di questa architettura tra i vari vantaggi offre la possibilità di **isolare parti del sistema**, evitando che gli errori si propaghino in modo incontrollato e che il sistema nel suo complesso vada in crash. Ogni sistema opera in modo indipendente e autonomo, con le proprie risorse e la propria logica.

Lo snodo centrale di comunicazione dei servizi, il middleware, è in grado di gestire le comunicazioni tra i vari microservizi e di garantire che il sistema sia in grado di funzionare anche in caso di guasti. Purtroppo non esiste alcun progetto senza alcun tipo di **single point of failure**, o almeno non con le risorse a disposizione di noi studenti, quindi all'interno di questo servizio la gestione degli errori dei servizi con cui comunica è gestita, ma di fatto, nel caso in cui il middleware smetta di funzionare, non avendo nessun altro servizio che possa fare da garante per il suo corretto funzionamento, tutto il sistema non permetterà, ad esempio, di giocare a nessun gioco. Questo perché il middleware è il cuore pulsante del sistema; avendo disponibilità di risorse e abbastanza conoscenze in un ambito reale, questo problema ha soluzioni:

- **Backup del middleware:** avere dei backup dei processi che si occupano di gestire le partite, ma questo significherebbe andare a modificare inevitabilmente la struttura già esistente del servizio.
- **Ridondanza del middleware:** avere molteplici istanze del middleware, preferibilmente 3, di cui una principale e le altre 2 in copia, facendo in modo che ogni azione sul middleware principale venga replicata sulle altre due istanze e che quindi, in caso di guasto hardware, le altre copie possano rispondere alle richieste. Se si utilizza questa soluzione bisogna prestare attenzione alla consistenza delle copie. La non consistenza potrebbe addirittura peggiorare la situazione. Per questo motivo è consigliato averne più di una, in modo tale da controllarne la consistenza.

Scalabilità

Nel caso del middleware, la scalabilità è un altro modo per chiamare la duplicazione, ma questo si applica soltanto al middleware, in quanto gli altri microservizi sono realmente scalabili. Ogni microservizio è in grado di scalare in modo indipendente, in base alle proprie esigenze, e questo permette di avere un sistema che si adatta in modo dinamico alle richieste degli utenti. Nella prossima sezione verrà approfondito come il deployment con Docker permetta di scalare i microservizi. L'implementazione di un sistema di load balancing viene fornita "gratuitamente" da Kubernetes, che permette di bilanciare il carico tra i vari nodi del cluster. Qualora il progetto maraffa-online dovesse avere un numero di utenti molto elevato, si potrebbe pensare di scalare il sistema in modo orizzontale, aggiungendo nuovi nodi al cluster, e tramite la dovuta configurazione, Kubernetes si occuperebbe di bilanciare il carico tra i vari nodi dei servizi scalabili.

Docker

Suddividere un progetto monolitico in servizi a sé stanti permette di dockerizzare comodamente ogni servizio e questo consente di avere un sistema facilmente deployabile, versionabile e portabile. Uno dei requisiti fondamentali del progetto era riuscire a dockerizzare interamente il sistema. Nel caso in cui si volesse fare un deploy su un server remoto, sarebbe sufficiente avere Docker installato, accedere al docker-compose del progetto, settare le corrette variabili d'ambiente e poi il progetto opererebbe in totale autonomia. Inoltre, utilizzare uno stack permette anche di gestire eventuali failure dei servizi in modo più semplice, poiché Docker consente di avere un sistema fault-tolerant. Se un servizio dovesse andare in crash per qualche motivo inspiegabile, Docker si occuperebbe di riavviarlo in modo autonomo. L'utilizzo di uno stack permette anche di scalare i servizi in modo più semplice, poiché basta settare il numero di repliche desiderate per un servizio e Docker si occuperà di creare le repliche e di bilanciare il carico tra di esse. Questo è un vantaggio nel caso in cui si voglia scalare il sistema in modo orizzontale, ovvero aumentando le istanze di un singolo servizio.

3.3.2 Struttura API Gateway

Il pattern di progettazione utilizzato per la realizzazione del middleware è quello dell'API Gateway, che permette di creare un'interfaccia unica per l'accesso ai vari servizi del sistema.

Nel nostro progetto, il client effettua semplici chiamate verso il middleware, che nei casi necessari effettua chiamate verso altri servizi per svolgere una determinata operazione. Queste chiamate non sono obbligatoriamente dirette verso il servizio interessato, ma possono anche essere routine che effettuano molteplici chiamate per ottenere il risultato desiderato. Implementare queste chiamate direttamente nel front-end avrebbe reso il codice molto più complesso e difficile da mantenere, e la gestione di partite multiple sarebbe stata molto più difficile.

Non è stato implementato nessun meccanismo di discovery per la gestione di chiamate verso gli altri servizi, in quanto tutti presenti sullo stesso server e nello stesso stack di Docker, quindi facilmente individuabili sulla rete virtuale creata da Docker.

Si è seguito un approccio modulare nello sviluppo di questo componente che, in un primo step, conteneva soltanto la logica di gestione delle partite, al quale sono stati aggiunti i moduli per la comunicazione con il servizio degli utenti e della business logic. Ogni modulo segue la stessa struttura, ossia una classe denominata controller che si occupa esclusivamente della dichiarazione delle rotte HTTP e che a sua volta contiene un service. Ogni servizio contiene poi le logiche di calcolo e crea un JSON di risposta che viene passato al controller che lo invia come risposta HTTP, questo per poter separare completamente e rispettare il principio di single responsibility.

Nel modulo contenente le logiche di gioco è stato necessario aggiungere un ulteriore strato alla struttura controller-service, data la struttura multi-agente della gestione delle partite che non permetteva una separazione netta. Quindi è stato aggiunto, tra il controller e il service, un componente ibrido per poter comunque avere servizi indipendenti.

Questa separazione, oltre a essere una buona pratica, è stata scelta per poter effettuare del testing comodamente soltanto sui servizi, testando quindi solamente la parte relativa all'elaborazione di una risposta.

3.3.3 WebSockets

L'interazione con il frontend avviene prevalentemente tramite il protocollo HTTP, ma per la gestione di alcuni aspetti in tempo reale sono state introdotte le WebSocket. Questo componente è stato creato, ad esempio, per la gestione delle chat di gioco, in modo che i giocatori possano comunicare tra loro in tempo reale. Ogni client, istanziato sul computer degli utenti, effettua una connessione WebSocket con il middleware utilizzando come endpoint il proprio UUID, che viene assegnato automaticamente all'avvio, e lo username del giocatore, dopo aver effettuato il login. Il middleware contiene al proprio interno una mappa di tutti i client e giocatori connessi con la relativa WebSocket e una classe che espone due metodi per inviare messaggi a un singolo giocatore o effettuare un messaggio broadcast a tutti i giocatori connessi.

Il modulo di game si occupa di gestire le partite, quindi è l'unico che può sapere quali giocatori devono ricevere determinati messaggi relativi a una singola partita. Per utilizzare il servizio di WebSocket, è stata implementata una semplice interfaccia all'interno dell'agente di gioco con la quale si effettuano chiamate "onEvent". Ogni volta che un particolare evento si verifica, un'invocazione a queste chiamate permette di inviare messaggi alle WebSocket dei giocatori interessati.

```
1 package game;
2
3 import game.service.User;
4
5 public interface IGameAgent {
6
7     void onCreateGame(User user);
8
9     void onNewRound();
10
11     void onJoinGame(User user);
12
13     void onStartGame();
14
15     void onCheckMaraffa(int suit, String username);
16
17     void onPlayCard();
18
19     void onTrickCompleted(Trick latestTrick);
20
21     void onMessage();
22
23     void onEndRound();
24
25     void onEndGame();
26
27     void onNewGame(String newGameID);
28
29     void onChangeTeam();
30
31     void onMakeCall(final Call call);
32 }
```

Listing 3.1: Interfaccia per gestire chiamate websocket

Capitolo 4

Implementazione

4.1 Componenti

Entrambi questi servizi sono stati realizzati con lo stack NestJS, un framework per Node.js che semplifica la realizzazione di web server e API REST. Si occupano rispettivamente di gestire la logica di gioco e la gestione degli utenti che vogliono utilizzare il sistema, due core business del progetto.

4.1.1 Business Logic

Questo servizio mantiene al proprio interno tutte le regole del gioco e il calcolo dei punteggi. Ha svariati endpoint che permettono al middleware di avere il minor numero possibile di responsabilità, delegando al servizio di business logic le operazioni riguardanti il gioco.

Questa scelta è stata fatta per mantenere il middleware il più leggero possibile e per evitare che si occupi di operazioni non di sua competenza. Inoltre, in futuro sarà possibile aggiungere nuovi giochi modificando il meno possibile il middleware, che dovrà semplicemente occuparsi della gestione delle partite e comunicare tempestivamente con i servizi di business logic in base al gioco scelto.

4.1.2 User Service

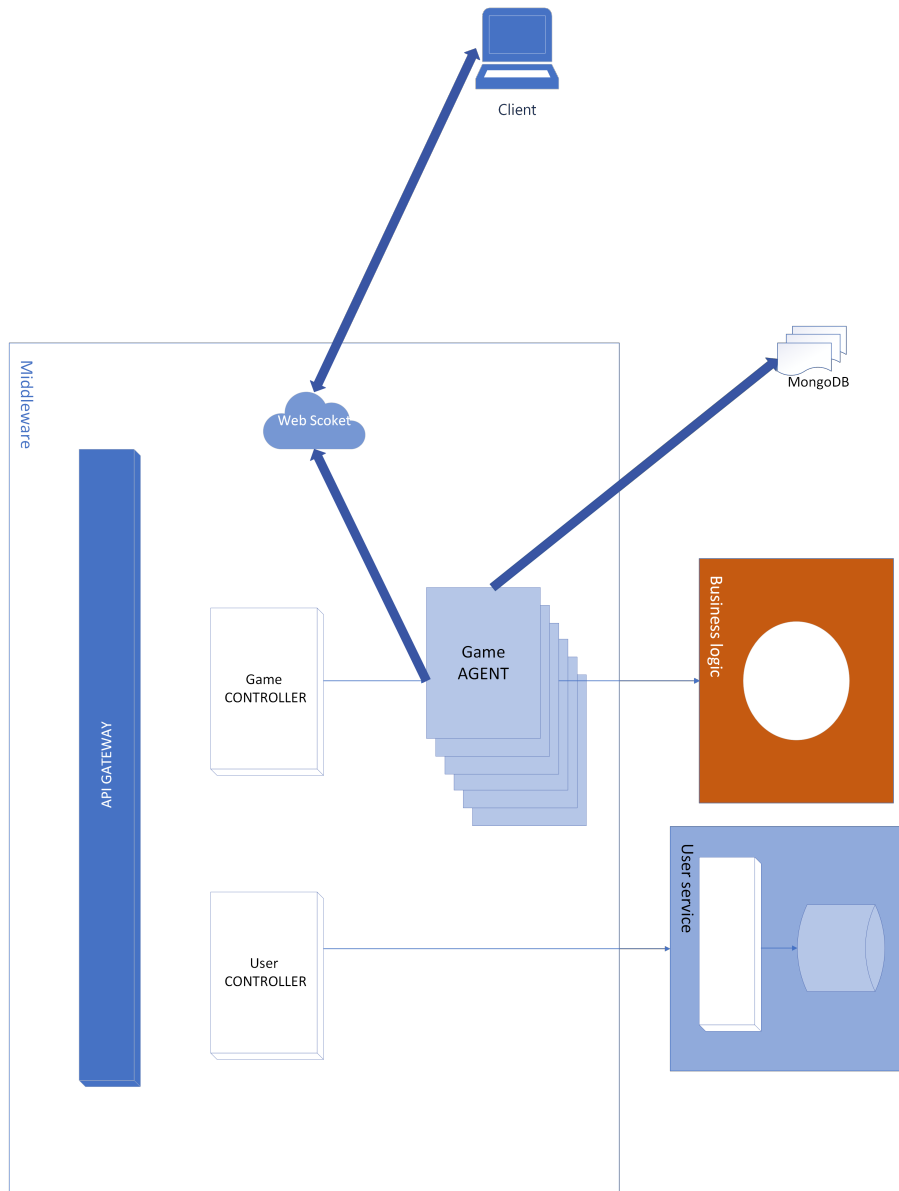
Questo servizio gestisce gli utenti, quindi la loro registrazione, autenticazione e gestione dei dati personali. Anche in questo caso, il middleware delega al servizio di gestione degli utenti le operazioni riguardanti gli utenti, per mantenere il middleware il più leggero possibile e per evitare che si occupi di operazioni non di sua competenza. Il core di questo servizio è dato dalla combinazione di NestJS e TypeORM, che permettono di gestire in modo semplice e veloce la persistenza dei dati e le operazioni CRUD su un database MySQL. Oltre alle informazioni degli utenti, il database contiene anche le informazioni riguardanti le partite giocate e le statistiche personali di ogni utente del sistema.

4.1.3 Front-End

Il front-end è stato realizzato con Angular e si occupa di gestire l'interfaccia grafica e la comunicazione con il middleware. Il progetto pilota di questo frontend è un template di una generica app in angular, questo velocizzato in parte fornendo una struttura e alcune best practices di angular di cui non si era a conoscenza. Sono state realizzate diverse pagine, tra cui la home page, la pagina di registrazione, la pagina di login, la pagina di gioco e la pagina di profilo utente. Utilizza API REST per comunicare con il middleware e ottenere i dati necessari

per il funzionamento del gioco, mentre utilizza WebSockets per la comunicazione in tempo reale con il middleware.

4.1.4 Middleware



Come già citato, il middleware è il cuore del sistema. L'architettura utilizzata per la gestione delle partite è quella di un multi-agente, in cui ogni agente si occupa di gestire una partita e di orchestrare la comunicazione con gli altri servizi, come ad esempio il front-end. Questo core è stato realizzato con Vertx, che è stato utilizzato anche per creare un web-server in Java.

Inoltre, al middleware è collegato un database NoSQL su MongoDB, che viene aggiornato a ogni trick completato dai giocatori per creare uno storico delle partite e salvare le loro decisioni in gioco. Questa raccolta di dati sarà parte di uno sviluppo futuro, in cui, dopo aver raccolto una grande quantità di dati di partite, si potrà tentare di creare un modello di machine learning per sviluppare un'intelligenza artificiale che possa giocare al posto di un giocatore umano.

4.2 Tecnologie

4.2.1 Swagger - OpenAPI

Uno dei requisiti fondamentali di questo progetto è stata la documentazione delle API. Per fare ciò è stato utilizzato Swagger, un framework open-source che permette di descrivere, produrre e consumare servizi web RESTful. Swagger permette di testare le API direttamente dalla documentazione, grazie a un'interfaccia grafica che consente di inviare richieste e visualizzare le risposte.

Per i servizi creati in Node.js è stata utilizzata una libreria ad hoc, in grado di generare la documentazione automaticamente tramite i corretti decoratori. Al contrario, per i servizi in Java non esiste alcuna libreria in grado di generare automaticamente la documentazione API a partire dai metodi esposti, quindi è stata creata manualmente. Grazie a un lavoro preliminare open-source svolto da Anup Saund sulla sua repository `Vertx auto swagger`, è stato possibile creare automaticamente un file HTML contenente la documentazione delle API esposte dai servizi in Java. Purtroppo, questo file non era in grado di aggiornarsi automaticamente in caso di modifiche alle rotte, quindi è stato necessario adattarlo per renderlo più flessibile e adatto al nostro progetto.

Vertx è una libreria davvero completa e ben strutturata. È stato possibile creare un Router in grado di gestire tutte le rotte del progetto in Java utilizzando una semplice classe astratta che ha definito tutti i metodi utilizzati nel progetto, tramite la quale si è potuto razionalizzare il codice e creare dinamicamente la funzionalità richiesta.

```
1 package httpRest;  
2  
3 import io.vertx.core.Handler;  
4 import io.vertx.core.http.HttpMethod;  
5 import io.vertx.ext.web.RoutingContext;  
6  
7 public interface IRouteResponse {  
8     HttpMethod getMethod();  
9     String getRoute();  
10    Handler<RoutingContext> getHandler();  
11 }
```

Listing 4.1: Semplice interfaccia per le rotte HTTP

La parte realmente complicata è stata fare in modo che le annotazioni presenti nel codice sorgente venissero lette e trasformate in un file JSON che rappresentasse correttamente la documentazione delle API. Dopo alcuni tentativi, è stato possibile automatizzare anche questa operazione.

```
1 final ImmutableSet<ClassPath.ClassInfo> modelClasses = ImmutableSet.<  
    ClassPath.ClassInfo>builder()  
2     .addAll(this.getClassesInPackage("game"))  
3     .addAll(this.getClassesInPackage("userModule"))  
4     .addAll(this.getClassesInPackage("BLManagement"))  
5     .addAll(this.getClassesInPackage("chatModule"))  
6     .build();
```

Listing 4.2: Aggiunta dei moduli delle classi contenenti rotte HTTP

Java permette di creare oggetti molto complessi, diversi dai JSON / object solitamente utilizzati in JavaScript. Era fondamentale poter adoperare questi schemi per avere una documentazione ricca e per poter testare velocemente le API senza utilizzare necessariamente un client come Postman.

Infine, per poter mostrare correttamente la documentazione, è stato necessario creare un file HTML che permettesse di visualizzare la documentazione in modo chiaro e ordinato. Questo file statico, presente nella repository del progetto, utilizza come fonte il file JSON di OpenAPI che viene rigenerato a ogni build del progetto, in modo da avere sempre la documentazione aggiornata.

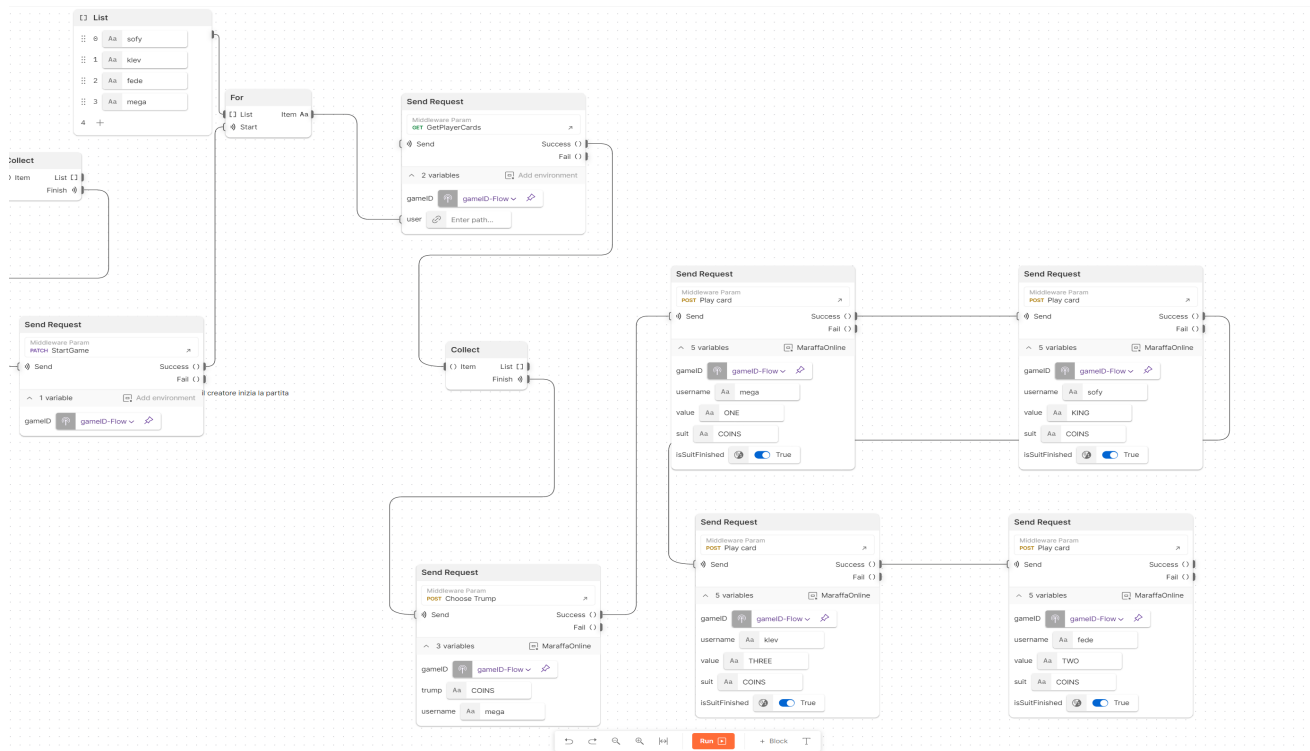
4.2.2 Postman

Swagger ha semplificato lo sviluppo delle API e il loro funzionamento, ma non è sostenibile per testare chiamate API successive che seguono un certo ordine o che necessitano di payload specifici. Postman è stato fondamentale per risolvere questo problema.

È stata creata una raccolta di richieste, comodamente esportate da Swagger senza alcun bisogno di riscrittura. Queste possono contenere payload parametrizzati grazie alle variabili di ambiente settabili una volta per tutte, come ad esempio il GamelD, che altrimenti, una volta creato, deve essere copiato e incollato in ogni richiesta successiva che lo richiede.

L'altro grande vantaggio di utilizzare Postman è stata la funzionalità Flows, che permette di creare un diagramma di flusso delle richieste e di salvare alcune risposte in variabili. Fino a quando non si è arrivati ad avere un frontend funzionante, questi Flows sono stati utili per trovare eventuali bug e avere ben chiaro il flusso di chiamate che sarebbero dovute essere implementate nel frontend, il che ha velocizzato lo sviluppo del frontend, seppur in minima parte.

È stato realizzato un flow che simulasse la creazione di una partita, l'iscrizione di 4 giocatori, l'avvio della partita e la simulazione di un turno completo di gioco. Questo ha permesso di testare in modo spedito molte delle funzionalità del sistema, e di trovare eventuali bug in modo altrettanto rapido.



4.2.3 Log automatici

Il servizio core del sistema è il middleware e per poterlo monitorare è stato necessario implementare un sistema di log automatici. Tenere traccia di tutte le operazioni è un compito molto complesso e avrebbe reso difficile un eventuale debug su un software in produzione che in caso di errore potrebbe riavviarsi e perdere definitivamente quell'errore. Grazie a Vertx, che offre "gratuitamente" un logger, e alla libreria LogBack, i log legati alle funzionalità più importanti sono trascritti su file .log in modo da poterli analizzare in caso di problemi.

Questo sistema poi si è evoluto grazie a una configurazione di LogBack in un file XML, tramite il quale è possibile personalizzare completamente il sistema di log, decidendo quali package loggare, in quali file trascriverli, scegliere il livello di log (info, debug, error, ecc.)

e attuare un sistema di log rotate. Secondo questa configurazione, i log oltre una certa dimensione vengono compressi e rinominati per non occupare troppo spazio, e si può anche impostare il numero di questi file compressi da mantenere.

```

1  <appender name="DEBUG_LOG" class="ch.qos.logback.core.rolling.
    RollingFileAppender">
2      <file>${DEBUG_FILE}</file>
3      <rollingPolicy class="ch.qos.logback.core.rolling.
        SizeAndTimeBasedRollingPolicy">
4          <fileNamePattern>${DEBUG_FILE}.%d{yyyy-MM-dd}.%i.gz</
            fileNamePattern>
5              <maxFileSize>900MB</maxFileSize>
6              <maxHistory>14</maxHistory>
7              <totalSizeCap>2GB</totalSizeCap>
8          </rollingPolicy>
9          <encoder>
10             <pattern>${LOG_PATTERN}</pattern>
11         </encoder>
12     </appender>
13
14
15     <!-- Il package game usera la configurazione del log di debug -->
16     <logger level="INFO" name="game" additivity="false">
17         <appender-ref ref="DEBUG_LOG_ASYNC"/>
18     </logger>

```

Listing 4.3: File configurazione Log automatici

Questo sistema si adatta perfettamente a un software dockerizzabile che ha montato un volume relativo alla cartella /log, che può anche essere un volume remoto, da cui è possibile consultare velocemente i log in caso di problemi.

```

1  FROM gradle:8.6.0-jdk17 AS build
2  COPY --chown=gradle:gradle . /home/gradle/src
3  WORKDIR /home/gradle/src
4  RUN gradle assemble
5  RUN gradle fatJar
6
7  FROM openjdk:19
8
9  RUN mkdir /app
10 RUN mkdir /app/log
11
12 COPY --from=build /home/gradle/src/app/build/libs/ /app/
13 COPY --from=build /home/gradle/src/app/log /app/log
14
15 EXPOSE 3003
16 ENTRYPOINT ["java", "-jar", "/app/Middleware.jar"]

```

Listing 4.4: Volume di log nel dockerfile l.13 da montare successivamente

4.2.4 Checkstyle, javadoc, github page

Dopo aver eseguito i controlli di stile sul codice sono stati trovati numerosi errori e warning, che sono stati corretti automaticamente da un plugin che ha sistemato ciò che Spotless non era riuscito a correggere. Il plugin in questione è OpenRewrite, che ha permesso di correggere molti errori e di risparmiare un lungo e noioso lavoro di correzione manuale.

```

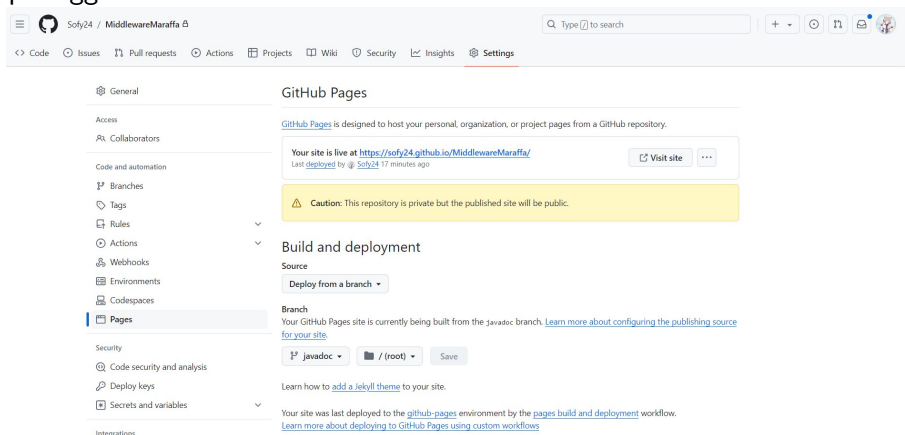
1 Changes have been made to app\src\main\java\userModule\UserController.java
  by:
2   org.openrewrite.staticanalysis.CodeCleanup
3   org.openrewrite.java.format.MethodParamPad
4 Changes have been made to app\src\main\java\userModule\UserService.java by
  :
5   org.openrewrite.staticanalysis.CodeCleanup
6   org.openrewrite.staticanalysis.NeedBraces
7 Changes have been made to app\src\test\java\integration\
  BusinessLogicTestIntegration.java by:
8   org.openrewrite.staticanalysis.CodeCleanup
9   org.openrewrite.java.format.MethodParamPad
10 Please review and commit the results.
11 Estimate time saved: 1h 24m

```

Listing 4.5: Output di OpenRewrite

Dopo aver corretto gli errori di stile, è stato generato il Javadoc per il codice, in modo da avere una documentazione automatica e aggiornata. Inoltre, è stato creato un sito web tramite GitHub Pages per avere una documentazione più user-friendly e facilmente accessibile.

Per fare ciò si è optato per l'utilizzo di una GitHub Action che, tramite condizioni come una push su determinati branch, genera automaticamente il Javadoc. Questa documentazione viene pubblicata su un branch 'javadoc', automaticamente creato, e tramite le impostazioni del repository, viene pubblicata su GitHub Pages, in modo da avere una documentazione sempre aggiornata e facilmente accessibile.



La documentazione del componente middleware sarà sempre reperibile all' indirizzo delle githubpages del progetto con il suffisso javadoc come ad esempio <https://sofy24.github.io/MiddlewareMaraffa/javadoc/>.

```

1 name: Deploy Javadoc
2
3 on:
4   push:
5     branches: [main, "develop", "style-doc"]
6
7 jobs:
8   publish:
9     runs-on: ubuntu-22.04
10    permissions:

```

```

11     contents: write # if you have a protection rule on your repository,
12     you'll need to give write permission to the workflow.
13     steps:
14       - name: Checkout
15         uses: actions/checkout@v4
16       - name: Deploy Javadoc
17         uses: MathieuSoysal/Javadoc-publisher.yml@v2.5.0
18         with:
19           GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
20           javadoc-source-folder: app/build/docs/javadoc
21           javadoc-branch: javadoc
22           java-version: 20
23           target-folder: javadoc # url will be https://<username>.github.
24           io/<repo>/javadoc, This can be left as nothing to generate javadocs in
25           the root folder.
26     project: gradle

```

Listing 4.6: Configurazione della GitHub Action per il Javadoc

4.2.5 Lettura variabili d'ambiente

Data la scelta di dockerizzare interamente i servizi, è stato necessario l'utilizzo di variabili d'ambiente. Questo ha permesso di creare un sistema di deploy molto flessibile, in cui è possibile cambiare l'indirizzo del servizio a cui connettersi semplicemente modificando il file di configurazione del Docker Compose. I servizi in NodeJS leggono il contenuto delle variabili d'ambiente tramite il modulo 'process.env', mentre i servizi in Java utilizzano 'System.getenv'. Questi comportamenti sono corretti per lo sviluppo locale e per ambienti di produzione in cui le variabili d'ambiente sono settate correttamente.

Per quanto riguarda invece lo sviluppo in CI, è stato necessario creare un file '.env.example' che contenesse tutte le variabili d'ambiente necessarie al funzionamento del servizio, in modo da poterle settare correttamente nel CI/CD, soprattutto per la fase di testing, e metterlo sulla repository. È importante tenere a mente che non dovrebbero mai essere caricati dati sensibili su strumenti di controllo di versione. In questo caso, non ci sono database o account cloud eventualmente raggiungibili dall'esterno.

4.3 Riflessioni sulla politiche di autovalutazione

- **l'elenco delle partite iniziate/finite deve rimanere aggiornato:** la reattività è stata garantita grazie alle WebSocket.
- **se un giocatore si disconnette la partita termina per tutti:** analizzando i casi pratici si è notato che quando un giocatore si disconnette è perchè non ha intenzione di rientrare, è pertanto inutile far attendere gli altri giocatori per una possibile riconnessione. Ogni giocatore viene notificato che la partita è stata terminata e, successivamente, vengono reindirizzati alla dashboard
- **connessione lenta o malfunzionamenti da parte di un giocatore:** il giocatore potrà rimanere sempre aggiornato sullo stato corrente della partita riaggiornando la pagina
- **un giocatore clicca sul tasto "esci":** allo stesso modo della disconnessione, verrà comunicato a tutti gli altri giocatori che la partita è terminata e verranno reindirizzati alla schermata principale

- **nel caso di un crash di un servizio si deve garantire che il servizio venga riavviato in automatico evitando che il sistema si blocchi:** se crasha un servizio docker ha una funzione builtin per riavviare in automatico il container

4.4 Mockup

Per soddisfare le aspettative degli utenti di Maraffa sono stati realizzati dei mockup. Grazie a questi si è riuscito a raccogliere opinioni per migliorare l'interfaccia grafica e l'usabilità del sito. Si riportano di seguito:

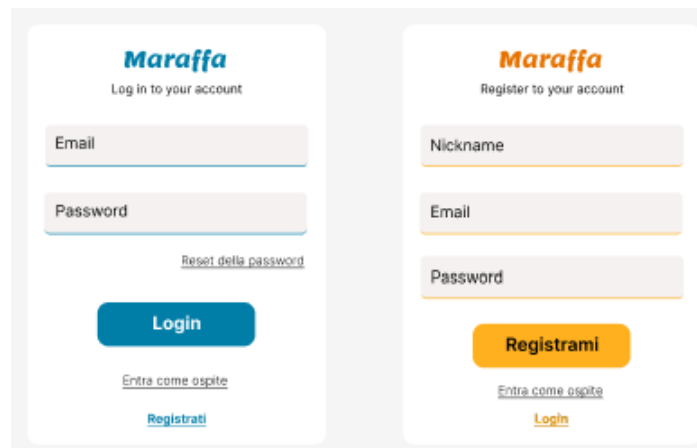


Figure 4.1: Mockup della schermata di Login e Registrazione

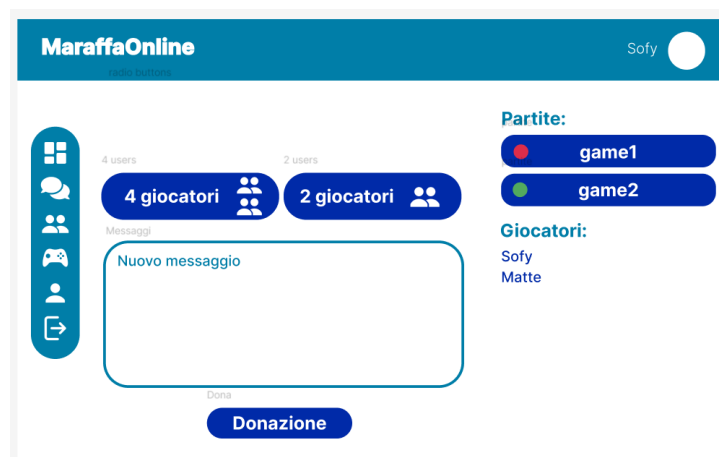


Figure 4.2: Mockup della dashboard

4.5 Screen

The mockup shows a web interface for 'MaraffaOnline'. At the top, there's a blue header with the site name and a user profile 'Sofy'. Below the header, a sidebar on the left contains icons for menu, chat, friends, game, and profile. The main content area is titled 'Profile' and features a user card for 'Sofia Tosi' with email 'email@.com'. To the right of the card is a 'Change password' form with three input fields: 'Current password', 'New password', and 'Confirm new password'. A blue 'Save' button is at the bottom of the form.

Figure 4.3: Mockup cambio password

The mockup shows a 'Game1' waiting room interface. The header is blue with 'MaraffaOnline' and a user profile 'Sofy'. Below the header, a sidebar on the left contains icons for menu, chat, friends, game, and profile. The main content area is titled 'Game1' and features a 'white in waiting room' status. A 'Menu' button is on the left. The central area displays 'In attesa di altri giocatori/in corso' and a list of participants: 'Team A' (Sofy, Klev) and 'Team B' (Matte). A 'Password:' field is at the bottom. 'JOIN' and 'PLAY' buttons are at the bottom.

Figure 4.4: Mockup della Waiting Room

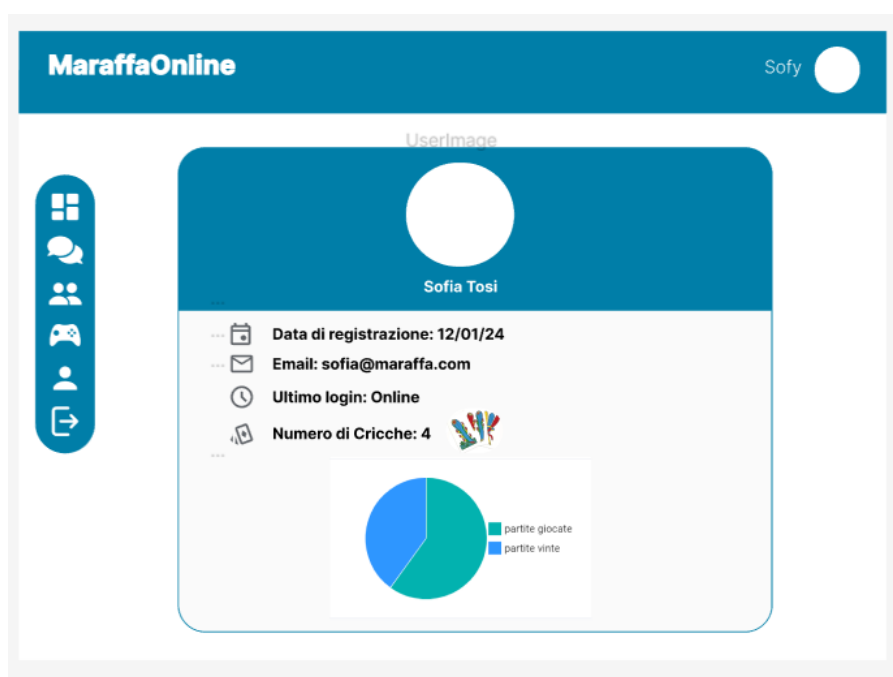


Figure 4.5: Mockup del profilo dell'utente

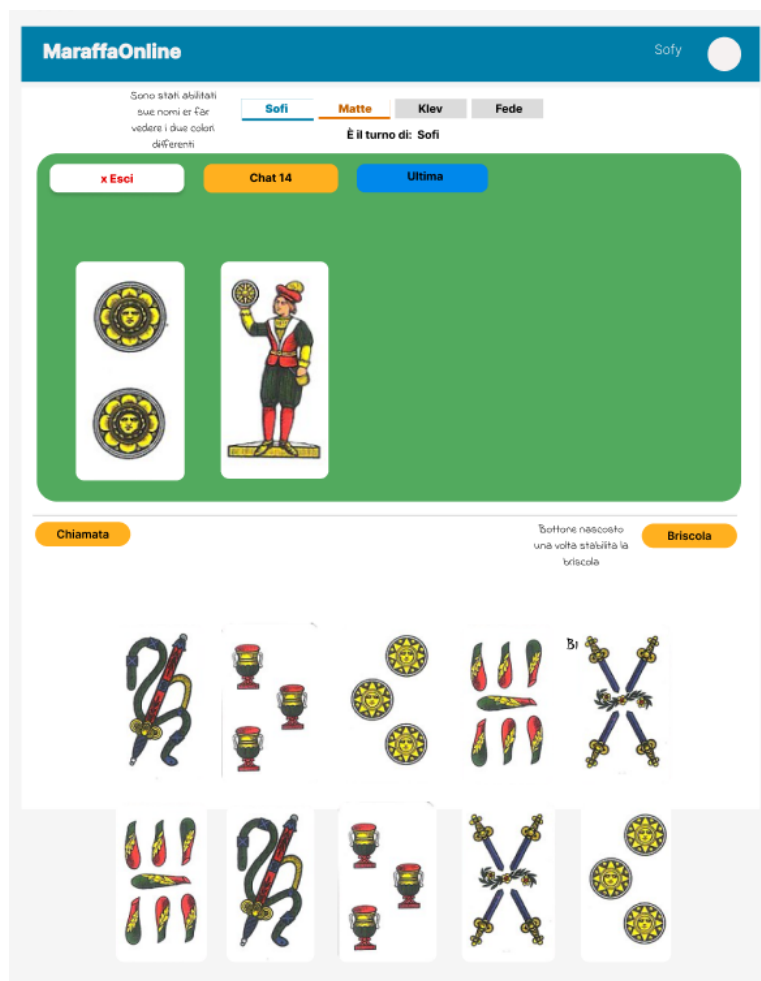


Figure 4.6: Mockup della schermata di gioco



The image shows a login and registration interface for 'MaraffaOnline'. The interface is centered on a light gray background. At the top, the logo 'MaraffaOnline' is displayed in a bold, blue, sans-serif font. Below the logo, the text 'Log in al tuo profilo' is written in a smaller, gray font. The form consists of two input fields: the first is labeled 'Nome Utente *' in blue and contains the text 'Elena'; the second is labeled 'Password *' in gray and contains four black dots. Below the password field is a blue button with the text 'Login' in white. Underneath the button, the text 'Entra come ospite' is displayed in a gray font. At the bottom of the form, the text 'Registrati' is displayed in a blue, underlined font.

MaraffaOnline

Log in al tuo profilo

Nome Utente *

Elena

Password *

....

Login

Entra come ospite

Registrati

Figure 4.7: Screen della schermata di Login e Registrazione



The image shows a registration form for 'MaraffaOnline'. The form is centered on a light gray background. It features three input fields: 'Email *' (light gray), 'Nome utente *' (light blue, containing 'Elena'), and 'Password *' (light blue, containing four dots). Below the fields is a gray 'Registrati' button, followed by the text 'Entra come ospite' and a red 'Login' link.

MaraffaOnline

Registrati

Email *

Nome utente *

Elena

Password *

....

Registrati

Entra come ospite

Login

Figure 4.8: Screen della schermata Registrazione

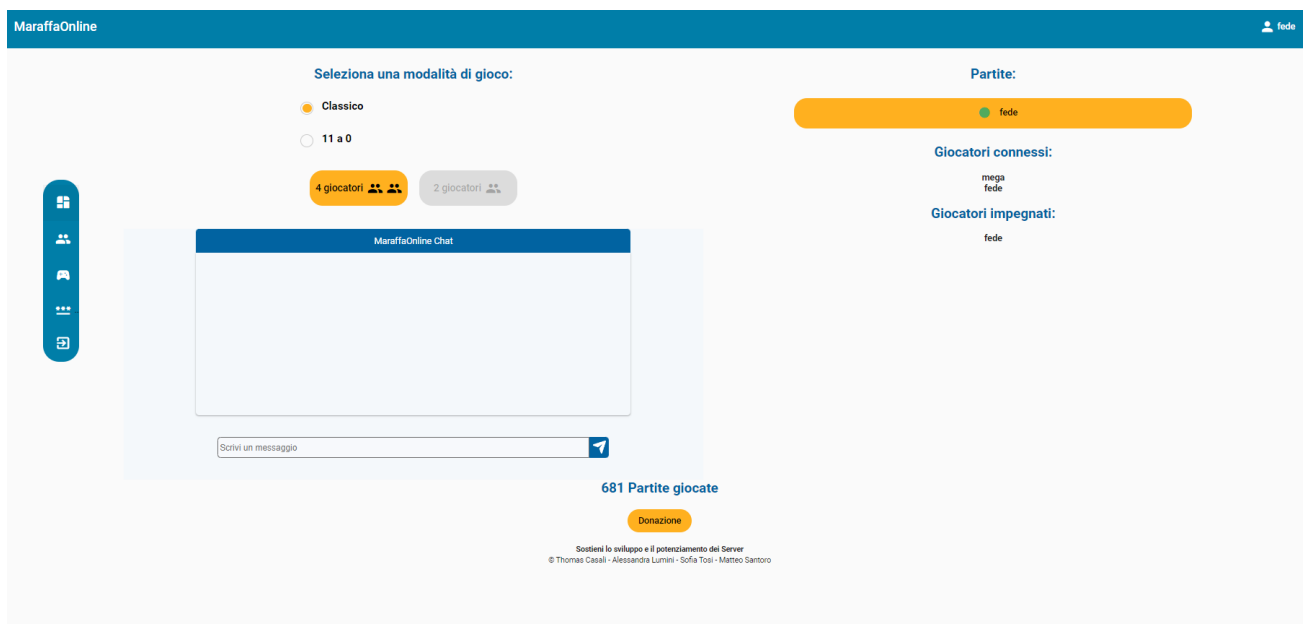


Figure 4.9: Screen Dashboard

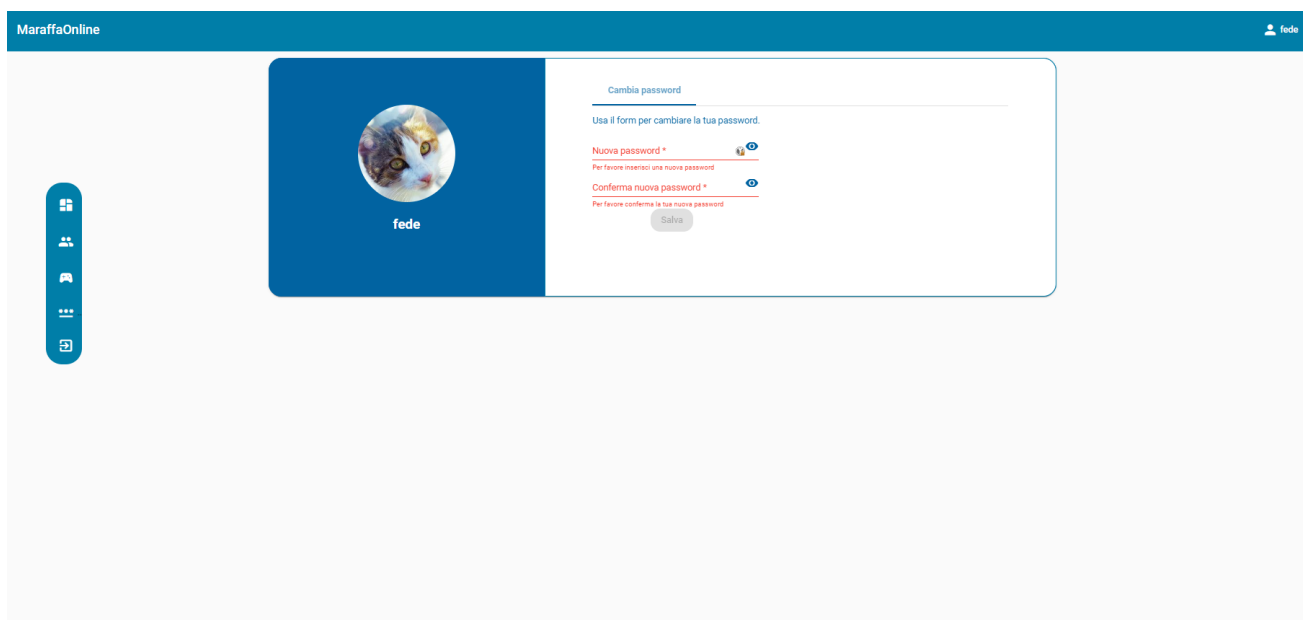


Figure 4.10: Screen cambio password



Figure 4.11: Screen della schermata waiting room

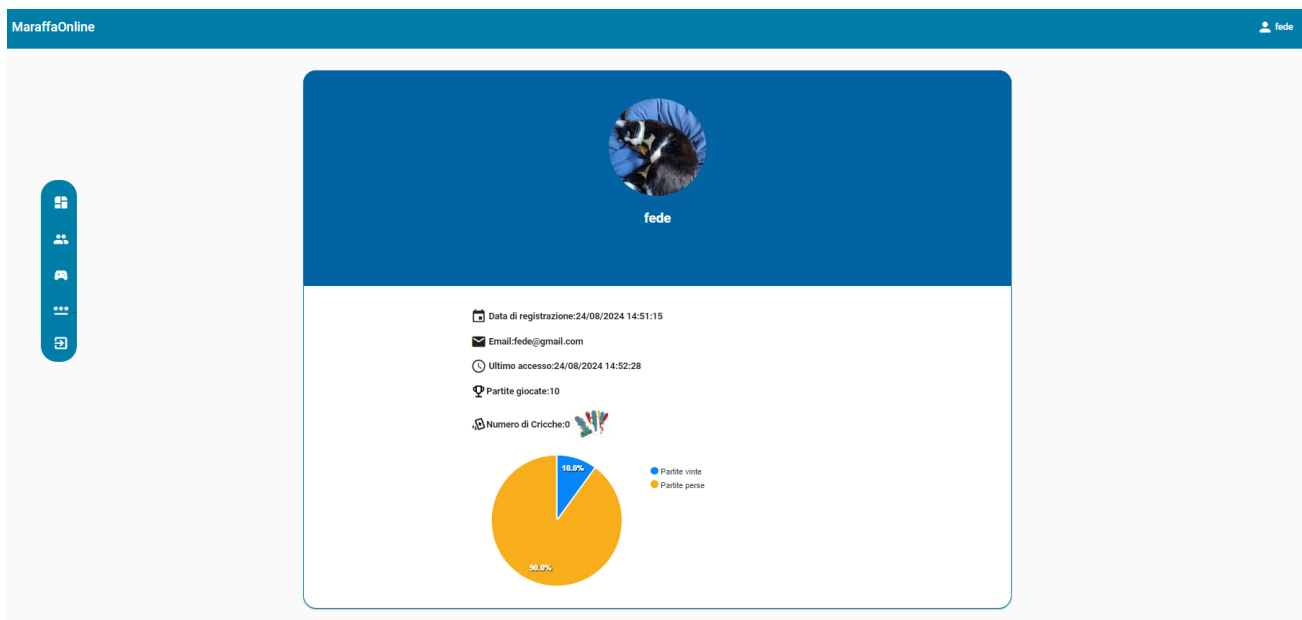


Figure 4.12: Screen del profilo dell'utente

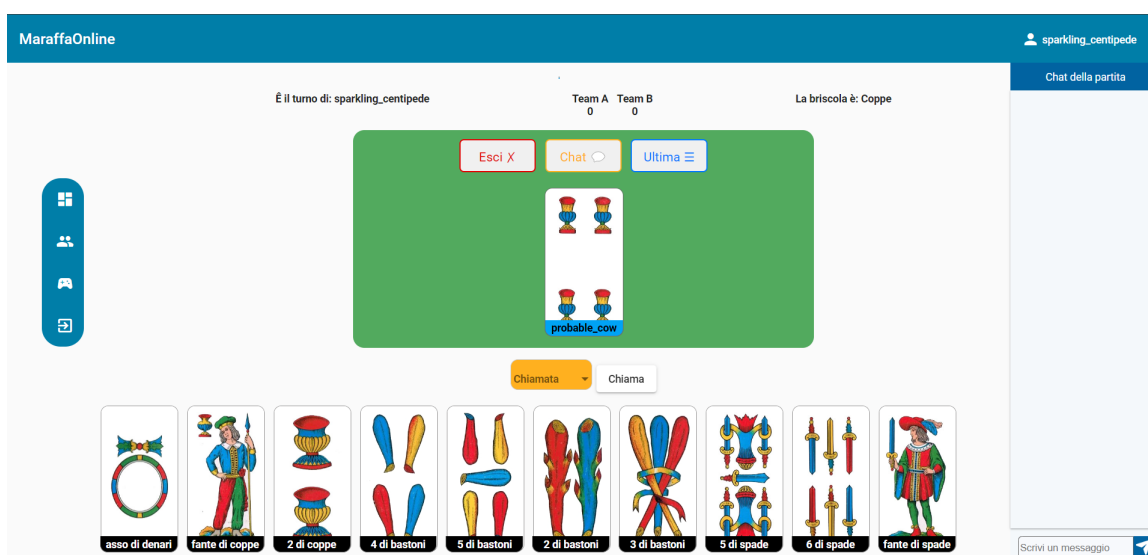


Figure 4.13: Screen della schermata di gioco

Capitolo 5

Interazioni

5.1 Frontend ↔ WebSocket

Come precedentemente citato, il middleware è stato implementato con WebSocket per poter comunicare in modo reattivo con il frontend. Queste comunicazioni frontend e middleware sono in buona parte scambi di messaggi tramite HTTP, ma in alcuni casi è necessario un canale di comunicazione bidirezionale. Uno degli esempi più "eclatanti" è la dashboard, in cui, in una prima implementazione, era stato utilizzato un polling per aggiornare i dati riguardanti le partite in corso per permettere agli utenti di partecipare. Immaginando uno scenario distribuito in cui ci sono N utenti contemporaneamente connessi al sistema, quella soluzione avrebbe sovraccaricato il sistema. Invece, tramite il canale bidirezionale delle WebSocket, è direttamente il middleware, che al proprio interno ha un servizio che mantiene una mappa dei client connessi, a notificare tramite un'operazione di broadcast tutti i client della creazione di una nuova partita.

5.1.1 Chat globale e privata

Un grande vantaggio dell'uso delle WebSocket sta nella facile implementazione delle chat. Questo sistema richiedeva l'impiego di due diverse chat per permettere la comunicazione tra utenti sia globalmente che all'interno di una singola partita. Anche in questo caso, il middleware si occupa di instradare i messaggi alle giuste chat, garantendo la privacy delle conversazioni. Il frontend, invece, si occupa di visualizzare i messaggi e di inviarli al middleware. La gestione della persistenza delle chat è stata implementata tramite l'uso del session storage del browser all'interno del middleware, permettendo di mantenere la chat anche in caso di refresh della pagina.

5.1.2 Reattività e aggiornamento in tempo reale

Tramite l'utilizzo delle WebSocket, è possibile ottenere un aggiornamento in tempo reale delle informazioni, senza dover fare polling continuo al server. Questa tecnologia è stata ampiamente utilizzata nella pagina della partita, permettendo ai client di ricevere aggiornamenti immediati, ad esempio sulle carte giocate da altri giocatori o su eventi scatenati da azioni specifiche, come l'abbandono di un giocatore, la gestione dei punteggi o la conclusione della partita stessa.

5.2 Stack Docker → reti interne e IP privati

Il sistema è stato progettato per essere eseguito su qualsiasi piattaforma e si è arrivati a creare uno stack completo di Docker Compose. Questo ha permesso anche la creazione di reti interne tra i container che vengono utilizzate per la comunicazione tra i servizi. È stato possibile assegnare IP privati ai container, permettendo di non esporre i servizi all'esterno e di mantenere un'architettura più sicura. Tra le variabili d'ambiente che vengono passate ai container c'è anche l'indirizzo del servizio a cui connettersi, e questo indirizzo è il nome del servizio all'interno della rete. Questo permette di non dover modificare il codice ogni volta che si cambia l'indirizzo del servizio a cui connettersi, ma basta modificare il file di configurazione del Docker Compose.

5.2.1 Nginx

Per quanto riguarda il frontend, è stato utilizzato Nginx per eseguire l'applicazione all'interno del container e per poterla raggiungere dall'esterno. Per fare ciò è stato necessario configurare Nginx tramite un file di configurazione che ha permesso anche di gestire un'operazione di reverse proxy per indirizzare le chiamate al backend.

```
1 server {
2     listen 80;
3
4     location / {
5         root /usr/share/nginx/html;
6         index index.html index.htm;
7         try_files $uri $uri/ /index.html;
8     }
9
10    location /api/ {
11        proxy_pass "http://${API_HOST}:${API_PORT}/";
12        proxy_http_version 1.1;
13        proxy_set_header Upgrade $http_upgrade;
14        proxy_set_header Connection 'upgrade';
15        proxy_set_header Host $host;
16        proxy_cache_bypass $http_upgrade;
17    }
18 }
```

Listing 5.1: Configurazione Nginx del front-end

La scelta di Nginx è stata fatta soprattutto per la sicurezza nel "mascherare" le chiamate fatte al backend, in modo da non esporre l'indirizzo del servizio all'esterno e poter superare eventuali problemi di CORS. L'indirizzo del backend è stato configurato anch'esso tramite variabili d'ambiente, in modo da poterlo cambiare facilmente in base all'ambiente in cui si trova il container, e queste variabili vengono sostituite a seconda dell'ambiente di deploy in cui l'applicativo Angular viene eseguito.

Capitolo 6

DevOps

6.1 DVCS

Come strumento di controllo di versione è stato scelto git, sono state adottate le seguenti best practices o git policy per garantire un flusso di lavoro coerente e prevenire conflitti di merge.

6.1.1 Struttura dei Branch

Una delle componenti principali della Git Policy è la gestione dei branch. La struttura tipica prevede almeno tre tipi di branch:

- **Master/Main:** Questo è il branch principale che contiene il codice di produzione. Ogni commit su questo branch dovrebbe essere stabile e pronto per il rilascio. Questo branch è stato protetto per evitare commit diretti, richiedendo quindi una revisione per ogni operazione di merge.
- **Develop:** Qui viene integrato il lavoro di sviluppo corrente. È il branch dove confluiscono le feature prima di essere preparate per il rilascio. Questo branch dovrebbe essere costantemente aggiornato e testato per assicurare che sia in uno stato pronto per la produzione.
- **Feature Branches:** Utilizzati per lo sviluppo di nuove funzionalità. Ogni feature branch deve derivare da 'develop' e, una volta completata la feature, viene reintegrato in 'develop' tramite una pull request.

6.1.2 Conventional commits

La pratica adottata per i commit è stata quella di effettuare commit frequenti e significativi.

In ogni repo è stato adottato un sistema standard per scrivere i commit: i conventional commit. In questo modo i commit risultano più chiari e facilmente leggibili. Riportiamo di seguito la nomenclatura usata:

- **fix:** per i commit che risolvono un bug
- **feat:** per i commit che aggiungono una nuova feature
- **refactor:** per i commit che migliorano il codice senza aggiungere nuove funzionalità
- **docs:** per i commit che riguardano la documentazione
- **style:** per i commit che riguardano la formattazione del codice

- **test:** per i commit che riguardano i test
- **ci:** per i commit che riguardano la Continuous Integration

Inoltre sono evidenziati i breaking changes per le modifiche non più compatibili con le versioni precedenti. La stessa nomenclatura viene utilizzata anche nel changelog.

6.2 Issue template

Il codice è open source ed è incoraggiata la collaborazione anche grazie alla presenza, in ogni repository, di template con i quali un utente può consigliare nuove feature, segnalare un bug o suggerire un'implementazione alternativa. Sono estremamente utili per ricevere feedback dagli utenti e monitorare gli eventuali bug.

6.3 Build automation

Abbiamo deciso di implementare ogni servizio in una repository diversa. La repository fornita è un contenitore di tutte le singole repository dei servizi. Per ogni repository è stata implementata una build automation specifica.

In questo modo si riesce a automatizzare i processi di build, test e deploy, ottenendo diversi vantaggi:

- si riduce drasticamente il tempo per la compilazione del codice
- ogni volta i passaggi verranno eseguiti allo stesso modo ottenendo, quindi, riproducibilità
- si identificano facilmente i problemi grazie al sistema di stampe a video che si ottiene durante l'esecuzione delle GitHub Actions
- minimizzazione degli errori umani
- garantisce il continuous deployment

6.3.1 Scheletro del workflow

I workflow utilizzati in questo progetto, pur utilizzando actions diverse specifiche per ogni servizio e linguaggio, seguono uno schema comune che ne facilita lo sviluppo e la manutenzione.

Ogni workflow di deployment è composto da tre job principali: build, test e deploy. Seguono una logica sequenziale e non parallela, in cui la fine di ogni job decreta l'inizio di quello successivo. Il collegamento tra i job è garantito da un sistema di cache, che permette di passare i dati tra i vari job migliorandone l'efficienza e la velocità. Questi job vengono attivati secondo alcune regole di progetto comuni alle repository, in linea con il workflow adottato. Questi ultimi non sono attivati ad ogni commit, ma solo in determinate circostanze, qui di seguito elencate:

- **Pull request su develop**
- **Release manuale su main pubblicata**

La struttura dei workflow è stata progettata per essere il più modulare possibile. Sono state utilizzate variabili d'ambiente per differenziare il deploy su develop e main, in modo da poter riutilizzare lo stesso workflow per entrambi i branch e produrre diverse immagini del software. Il nome dell'immagine pubblicata viene creato in modo parametrico, utilizzando il nome della repository, modificando i caratteri in minuscolo e aggiungendo la versione del software, che viene presa dal tag della release o impostata su stage o develop a seconda del branch che ha attivato il workflow.

6.3.2 UserManagement e BusinessLogic

Essendo le due repository estremamente simili e avendo entrambe un ambiente in Node.js, è stato implementato un workflow pressoché analogo. Le actions utilizzate sono quelle di Yarn per le varie fasi di build, test e deploy, e per la creazione della cache tra i vari job.

6.3.3 Middleware

Questa repository è stata implementata in Java, quindi il workflow è stato leggermente diverso. Per le varie fasi sono state utilizzate le action di Java e Gradle. Per il corretto funzionamento della action di setup e quindi delle successive, si è reso necessario, anche se non previsto e non consigliato, mantenere sul repository remoto il file 'gradle-wrapper.jar'. Per la fase di test, invece, è stata utilizzata un'action che permette di creare un container con un database di MongoDB, in modo da poter testare il codice che interagisce con un database, il che compone una delle funzionalità principali del servizio. Anche in questo caso, password e username vengono passati tramite i secrets.

6.3.4 Frontend

La repository di frontend è stata implementata in Angular, quindi il workflow è stato leggermente diverso. Per le varie fasi sono state utilizzate le action di Yarn per le fasi di build, deploy e per la creazione della cache tra i vari job. Non è stato necessario utilizzare la fase di test, poiché il frontend di un'applicazione non ha un sistema di unit test; la comunicazione con il middleware viene testata "manualmente".

6.4 Continuous integration

Per migliorare la qualità del software e velocizzare il processo di sviluppo, è stata implementata la Continuous Integration. Questa pratica permette di integrare il codice frequentemente, evitando possibili rischi d'integrazione e consentendo l'esecuzione di test automatici. Grazie al sistema di notifiche, inoltre, è possibile ricevere feedback immediato sullo stato delle Github Actions, per accertarsi che sia sempre tutto funzionante e non siano stati introdotti bug. Mantenendo il sistema continuamente monitorato, si migliora anche l'efficienza e la collaborazione tra i membri del team.

Capitolo 7

Validazione

7.1 Test

Per ogni servizio in questo sistema è stato adottato, ove possibile, il paradigma di programmazione TDD, cioè Test Driven Development. Questo approccio prevede che i test siano scritti prima del codice, in modo da guidare lo sviluppo e garantire che il codice prodotto soddisfi i requisiti specificati. Le operazioni di testing sono state implementate con Jest per quanto riguarda i servizi che usano Node.js, generando di fatto un report abbastanza comprensibile. Il componente middleware, scritto in Java, invece, generava risultati di test non così chiari. Per modificare questo comportamento, è stata introdotta una dipendenza nel build Gradle che permette di avere un report più dettagliato e comprensibile. La libreria `Test Logger`, tramite una piccola configurazione nel file `build.gradle` come qui riportata, ha soddisfatto le nostre esigenze.

```
1 import com.adarshr.gradle.testlogger.TestLoggerExtension
2 import com.adarshr.gradle.testlogger.TestLoggerPlugin
3 import com.adarshr.gradle.testlogger.theme.ThemeType
4
5 testlogger {
6     theme = ThemeType.MOCHA
7     showExceptions = true
8     showStackTraces = true
9     showFullStackTraces = false
10    showCauses = true
11    slowThreshold = 2000
12    showSummary = true
13    showSimpleNames = false
14    showPassed = true
15    showSkipped = true
16    showFailed = true
17    showOnlySlow = false
18    showStandardStreams = false
19    showPassedStandardStreams = true
20    showSkippedStandardStreams = true
21    showFailedStandardStreams = true
22    logLevel = LogLevel.LIFECYCLE
23 }
```

Listing 7.1: Test logger

L'architettura dei componenti si può riassumere in questo modo:

- **Classi controller:** che hanno la funzione di esporre le rotte e di contenere al loro interno classi di servizio.
- **Classi di servizio:** che contengono la logica di business.

I test vengono quindi eseguiti sulle classi di servizio, andando a testare la logica di business e non le funzioni. Questo approccio permette di avere una copertura maggiore del codice e di garantire che la logica di business sia corretta. Un esempio di test su una classe di servizio, prendendo come esempio il componente `userManagement`, rende chiaro il concetto.

Questo componente di fatto svolge due funzioni ben distinte:

- Collegarsi a un database MySQL e fare delle operazioni CRUD su tabelle di utenti e le loro rispettive statistiche.
- Gestire la registrazione e l'autenticazione degli utenti.

La parte di testing quindi non va assolutamente a verificare che la connessione al database funzioni, ma che le operazioni CRUD siano corrette e che la registrazione e l'autenticazione degli utenti avvengano correttamente, così come l'aggiornamento delle statistiche.

Avendo scelto un'architettura a microservizi, è stato necessario testare anche il corretto funzionamento delle comunicazioni tra i servizi. Per fare ciò, è stato necessario creare dei test di integrazione che verificassero il corretto funzionamento delle API esposte dai servizi. Questi test sono stati implementati nel componente `middleware` che, come sicuramente si ricorderà, ha il compito di mettere in comunicazione i servizi tra di loro e con il frontend.

7.1.1 Testing asincrono

Durante la scrittura dei test all'interno del servizio in Java, alcune operazioni erano asincrone e quindi è stato necessario attendere che queste operazioni terminassero prima di poter eseguire i test e validarli. Per fare ciò, la normale struttura di test fornita da `Vertx` in un primo tentativo non era sufficiente. La classica struttura dei test è la seguente:

```

1 @Test
2 public void createGameTest(final VertxTestContext context) {
3     final JsonObject gameResponse = this.gameService.createGame(
4         MARAFFA_PLAYERS, TEST_USER, EXPECTED_SCORE, GAME_MODE.toString(),
5         PASSWORD);
6     Assertions.assertEquals(UUID_SIZE, gameResponse.getString(Constants.
7         GAME_ID).length()); // Assuming UUID is 36
8     context.completeNow();
9 }

```

Listing 7.2: Standard `Vertx` test

Per risolvere questo problema sono stati utilizzati:

- il decoratore fornito da `JUnit Jupiter` `@Timeout`, che permette di aspettare un tempo definito prima di fallire automaticamente il test, per ovviare al problema di asincronicità con servizi esterni al componente stesso che non "rispondono alla chiamata".
- l'utilizzo asincrono del `VertxTestContext`, che permette di completare il test solo quando tutte le operazioni asincrone sono state completate.

- una struttura di risposta nei servizi con cui il middleware comunica, che permette di individuare errori tramite la chiave "error" in caso di fallimento contenuta all'interno del JSON di risposta. Il test non può accedere alla response di una chiamata HTTP che normalmente avrebbe uno status code 200 in caso di successo e quindi si è ricorso a questo espediente per individuare errori nei dati inviati al servizio.

Come è possibile vedere nel test riportato di seguito, seguendo questi accorgimenti è possibile testare correttamente le operazioni asincrone.

```

1 @Timeout(value = 10, unit = TimeUnit.SECONDS)
2 @Test
3 public void testgetShuffledDeckOK(final VertxTestContext context) {
4     final JsonObject gameResponse = this.gameService.createGame(4,
5         TEST_USER, 41,
6         GameMode.CLASSIC.toString(), PASSWORD);
7     this.businessLogicController
8         .getShuffledDeck(UUID.fromString(gameResponse.getString(
9             Constants.GAME_ID)), 4)
10        .whenComplete((res, err) -> {
11            context.verify(() -> {
12                assertNull(res.getString("error"));
13                context.completeNow();
14            });
15        });
16 }

```

Listing 7.3: Vertx test asincrono

7.2 Code quality e linting

Durante lo sviluppo dei servizi è stata adottata una politica di code quality e linting per garantire la coerenza e la leggibilità del codice.

Per i servizi in Node.js, sono stati utilizzati ESLint assieme a Prettier. ESLint è uno strumento di analisi statica del codice che aiuta a identificare e correggere gli errori di codice, le pratiche non ottimali e le violazioni dello stile di codice. Questi strumenti necessitano soltanto di un file di configurazione che è presente nelle repository e che i software per la scrittura di codice, come ad esempio VSCode, possono adoperare.

Il componente in Java ha adottato Checkstyle, un tool di analisi statica del codice che aiuta a garantire che il codice Java rispetti uno standard di codifica predefinito. Anche in questo caso è stato sufficiente aggiungere una dipendenza nel build.gradle per ottenere un report dettagliato e comprensibile ed un file di configurazione che è stato inserito nella repository. Per automatizzare il processo di formattazione del codice il più possibile è stata aggiunta la libreria Spotless. L'aggiunta di queste dipendenze nel file build.gradle è stata sufficiente per garantire la formattazione del codice e la sua coerenza.

```

1 spotless {
2     java {
3         importOrder() // standard import order
4         removeUnusedImports()
5         googleJavaFormat() // has its own section below
6         eclipse() // has its own section below
7     }
8 }

```

```
9 checkstyle {
10     toolVersion = "8.44" // Versione di Checkstyle
11     configFile = file("${rootDir}/config/checkstyle/checkstyle.xml") //
    Configurazione di Checkstyle
12     // showViolations = true
13 }
```

Listing 7.4: Code quality

Capitolo 8

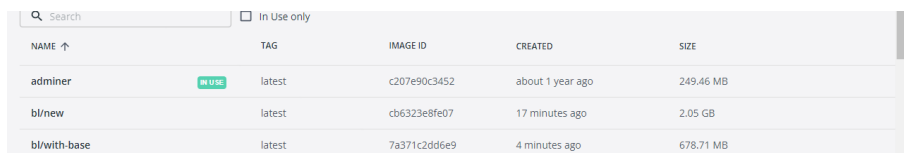
Deployment

8.1 Containerization

Data l'architettura del progetto in microservizi, usare Docker per creare un'immagine per ogni servizio è stata una scelta naturale. L'obiettivo finale è avere un sistema facilmente deployabile e scalabile, eseguibile velocemente in un ambiente generico, con uno stack di container Docker che racchiuda tutto il sistema. Per fare ciò è stato necessario creare un Dockerfile per ogni servizio, in modo da poter creare un'immagine durante la fase di deploy della continuous integration.

La **strategia di containerizzazione** per tutte le immagini create è stata quella di utilizzare un'immagine base di Alpine, in modo da avere immagini leggere e veloci da scaricare. Per ridurre la dimensione delle immagini, si è utilizzata un'immagine di sviluppo per compilare il codice e un'immagine di produzione per eseguire il codice compilato, copiando solo i file necessari.

Figure 8.1: Comparazione tra le immagini multi-stage e le immagini tradizionali



The screenshot shows a table of Docker images. At the top, there is a search bar and a checkbox labeled 'In Use only'. The table has columns for NAME, TAG, IMAGE ID, CREATED, and SIZE. Three images are listed: 'adminer' (249.46 MB), 'bl/new' (2.05 GB), and 'bl/with-base' (678.71 MB). The 'bl/new' image is marked as 'In Use'.

NAME ↑	TAG	IMAGE ID	CREATED	SIZE
adminer	latest	c207e90c3452	about 1 year ago	249.46 MB
bl/new	latest	cb6323e8fe07	17 minutes ago	2.05 GB
bl/with-base	latest	7a371c2dd6e9	4 minutes ago	678.71 MB

8.1.1 NodeJS

I servizi UserManagementMaraffa e BusinessLogic sono stati containerizzati utilizzando un'immagine di Node.js. Nell'esempio sotto, si può notare la distinzione tra i diversi stage.

```
1 FROM node:20-alpine as base
2 WORKDIR /app
3 COPY yarn.lock package.json /app/
4 RUN yarn install
5 COPY . /app
6 RUN yarn build
7
8 FROM node:20-alpine
9 WORKDIR /app
10 COPY --from=base /app/package.json /app/package.json
11 COPY --from=base /app/node_modules /app/node_modules
12 COPY --from=base /app/dist /app/dist
13 EXPOSE 3000
14 CMD ["node", "dist/main.js"]
```

Listing 8.1: Dockerfile delle immagini NodeJS

8.1.2 Java

La containerizzazione del middleware ha richiesto alcuni passaggi aggiuntivi rispetto ai servizi in Node.js. È stato necessario adottare immagini diverse per gli stage di build e produzione, quindi usare un'immagine di Gradle per la build, nella quale eseguire i comandi di Gradle, e un'immagine di OpenJDK per l'esecuzione del JAR prodotto dalla build.

Per la compilazione con Gradle, nonostante non sia una pratica corretta, è stato necessario mantenere nella repository il file gradle-wrapper.jar, in quanto non era possibile scaricarlo durante la build all'interno delle GitHub Actions. La build non utilizza il classico comando di Gradle per generare il file .jar, ma un task personalizzato che si occupa di generare il fatJar del servizio, poiché le dipendenze non venivano gestite correttamente all'interno del normale file .jar.

```
1 tasks.register<Jar>("fatJar") {
2     archiveBaseName.set("Middleware")
3     manifest {
4         attributes["Main-Class"] = "server.Main"
5     }
6     from(sourceSets.main.get().output)
7     dependsOn(configurations.runtimeClasspath)
8     from({ configurations.runtimeClasspath.get().filter { it.name.endsWith
9         ("jar") }.map { zipTree(it) } })
10    dependsOn("compileJava")
11    duplicatesStrategy = DuplicatesStrategy.EXCLUDE // Puoi utilizzare
12    // altre strategie come DuplicatesStrategy.WARN per avvisare ma non
13    // fermare la build
14 }
```

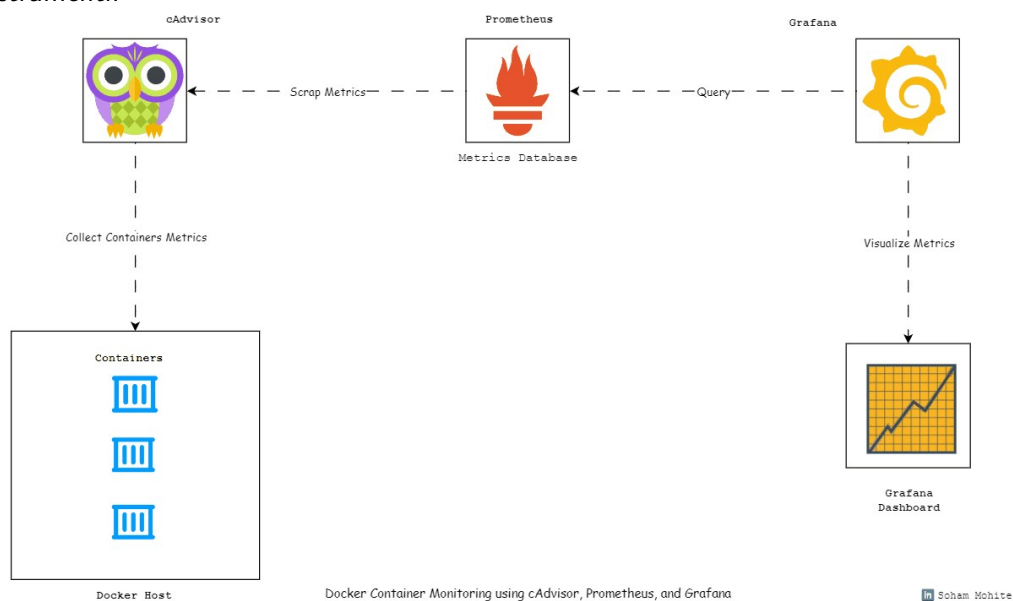
Listing 8.2: Task del fatJar da includere nel container

8.1.3 Angular

La containerizzazione del frontend ha richiesto l'utilizzo di un'immagine di Node.js per lo stage di build, mentre per l'esecuzione è stata utilizzata un'immagine di Nginx. Come citato nella sezione 5.2.1 Nginx serve per eseguire l'applicazione all'interno del container e per poterla raggiungere dall'esterno.

8.2 Monitoraggio

In un progetto composto da microservizi deployati con container, il monitoraggio è essenziale per garantire il corretto funzionamento e la performance ottimale del sistema. Per realizzare un monitoraggio efficace dei container, sono stati impiegati i container di Grafana, Prometheus e cAdvisor. Si ringrazia il lavoro di [Soham Mohite](#) per la configurazione e l'integrazione di questi strumenti.



8.2.1 Grafana

Grafana è uno strumento open-source per la visualizzazione e l'analisi delle metriche raccolte. Viene utilizzato per creare dashboard personalizzate che mostrano lo stato e le performance dei microservizi. Con Grafana, è possibile configurare alert che notificano immediatamente eventuali problemi nel sistema, permettendo una risposta rapida e mirata.

8.2.2 Prometheus

Prometheus è un sistema di monitoraggio e di allarme progettato per raccogliere e memorizzare metriche in serie temporali. È stato configurato per raccogliere metriche dai container dei microservizi e da cAdvisor. Prometheus esegue la raccolta dei dati a intervalli regolari e li memorizza in un database time-series, rendendoli disponibili per l'analisi e la visualizzazione in Grafana.

8.2.3 cAdvisor

cAdvisor (Container Advisor) è uno strumento che fornisce informazioni sulle risorse utilizzate dai container, come CPU, memoria, rete e disco. È stato integrato con Prometheus per raccogliere e esportare le metriche dei container. cAdvisor offre una visione dettagliata delle performance di ogni container, aiutando a identificare e risolvere problemi di utilizzo delle risorse.

8.2.4 Integrazione e Configurazione

L'integrazione di Grafana, Prometheus e cAdvisor ha permesso di creare un sistema di monitoraggio completo. I container di cAdvisor raccolgono le metriche di utilizzo delle risorse dai container dei microservizi e le esportano a Prometheus. Prometheus memorizza queste metriche e le rende disponibili per la visualizzazione in Grafana. Le dashboard di Grafana sono configurate per mostrare le metriche chiave e fornire un'analisi dettagliata dello stato del sistema.

La configurazione di questi strumenti è stata gestita tramite Docker Compose, permettendo una facile implementazione e scalabilità del sistema di monitoraggio. Le variabili d'ambiente e i file di configurazione sono stati impostati per garantire la corretta connessione e funzionamento dei container di monitoraggio.

```
1  cadvisor:
2    container_name: cadvisor
3    image: gcr.io/cadvisor/cadvisor:latest
4    ports:
5      - "8080:8080"
6    volumes:
7      - "/:/rootfs"
8      - "/var/run:/var/run"
9      - "/sys:/sys"
10     - "/var/lib/docker:/var/lib/docker"
11     - "/dev/disk:/dev/disk"
12    privileged: true
13    devices:
14      - "/dev/kmsg"
15
16  prometheus:
17    container_name: prometheus
18    image: prom/prometheus:latest
19    ports:
20      - "9090:9090"
21    volumes:
22      - "./prometheus.yml:/etc/prometheus/prometheus.yml"
23    privileged: true
24    depends_on:
25      - cadvisor
26
27  grafana:
28    container_name: grafana
29    image: grafana/grafana:latest
30    ports:
31      - "3000:3000"
32    environment:
33      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
34      - DS_PROMETHEUS=prometheus
35    volumes:
36      - "grafana-data:/var/lib/grafana"
```

```
37     - "./datasources.yml:/etc/grafana/provisioning/datasources/
    datasources.yml"
38     - "./dashboard.json:/var/lib/grafana/dashboards/dashboard.json"
39     - "./default.yml:/etc/grafana/provisioning/dashboards/default.yml"
40 privileged: true
41 depends_on:
42     - prometheus
```

Listing 8.3: Esempio di configurazione Docker Compose

Con questa configurazione, il sistema di monitoraggio offre una visione completa delle performance e dello stato dei microservizi, permettendo di mantenere un'operatività ottimale e di intervenire prontamente in caso di anomalie.

8.2.5 Esecuzione del sistema

L'esecuzione del sistema può essere avviata tramite un unico file `docker-compose.yml`. Con l'ambiente configurato correttamente, il sistema è in grado di operare autonomamente dopo l'esecuzione del comando `docker compose up`.

Per l'esecuzione dei singoli servizi, sono necessari alcuni accorgimenti specifici. Il middleware e la business logic possono essere avviati in modo indipendente, poiché sono sempre gli altri servizi a interagire con loro. Lo User Service, invece, richiede un collegamento attivo a un database, mentre il frontend necessita di una serie di chiamate API e connessioni WebSocket al middleware (o a un componente che implementi le sue interfacce).

In ogni caso, ciascun servizio può essere eseguito, esteso o migliorato in modo indipendente rispetto agli altri, a condizione che vengano rispettate le interfacce di comunicazione condivise tra i servizi.

Capitolo 9

Conclusions and Future Work

9.1 Conclusioni

L'architettura basata su micro-servizi si è rivelata una scelta altamente vantaggiosa per il progetto, garantendo modularità, scalabilità e resilienza. Ogni micro-servizio è stato progettato per gestire specifiche funzionalità, e la loro integrazione avviene tramite API REST e websocket orchestrate da un middleware centrale, che facilita una comunicazione efficiente e ben strutturata tra le componenti del sistema. L'adozione di Docker ha semplificato ulteriormente il processo di deploy, offrendo un ambiente di esecuzione coerente e adattabile, capace di gestire le diverse esigenze di configurazione e scaling e affidabilità.

L'architettura basata su micro-servizi si è rivelata una scelta altamente vantaggiosa per il progetto, garantendo modularità, scalabilità e resilienza. Questo approccio consente una maggiore flessibilità nello sviluppo, permettendo ai team di lavorare in modo indipendente su componenti diverse, riducendo al contempo le interdipendenze e i conflitti di integrazione.

Quest'integrazione avviene tramite API REST e websocket orchestrate da un middleware centrale, che facilita una comunicazione efficiente e ben strutturata tra le componenti del sistema. Questo sistema intermediario, gestisce le richieste e il bilanciamento del carico, garantendo che i dati siano trasferiti in modo rapido tra i vari micro-servizi. Inoltre, l'uso di websocket consente una comunicazione bidirezionale in near real-time, migliorando la reattività dell'applicazione.

L'adozione di Docker ha semplificato ulteriormente il processo di deploy, offrendo un ambiente di esecuzione coerente e adattabile. Docker, infatti, permette di incapsulare ogni micro-servizio in un container, che include tutte le dipendenze necessarie, eliminando problemi legati alle differenze tra ambienti di sviluppo e di test. Questo approccio non solo facilita la distribuzione, ma migliora anche la portabilità delle applicazioni e la gestione delle risorse. Grazie a Docker, è possibile scalare orizzontalmente i micro-servizi in base alle necessità, rispondendo rapidamente alle variazioni del carico di lavoro e garantendo così un'elevata disponibilità e affidabilità del sistema.

Infine, l'architettura a micro-servizi, risulta essere altamente resiliente. In caso di fallimento di un singolo micro-servizio, il sistema può continuare a funzionare, con la possibilità di isolare e risolvere il problema senza impattare l'intera applicazione. Questo approccio modulare non solo migliora la robustezza del sistema, ma facilita anche l'implementazione di nuove funzionalità e l'aggiornamento di quelle esistenti, rendendo l'intero processo di sviluppo più agile e reattivo.