Piperchat

Alessandro Mazzoli

alessandro.mazzoli9@studio.unibo.it

Luigi Borriello

luigi.borriello2@studio.unibo.it

Manuel Andruccioli

manuel.andruccioli@studio.unibo.it

Tommaso Patriti

tommaso.patriti@studio.unibo.it

10 dicembre 2023

Il progetto consiste nella realizzazione di una piattaforma che permette la comunicazione fra gli utenti, ispirata a Discord[1].
Piperchat, mira a creare un ambiente di comunicazione ispirato a Discord, dove gli utenti possono interagire in varie forme. Offrirà la possibilità di registrazione e accesso tramite un sistema di login, consentendo agli utenti di stabilire e gestire connessioni sociali attraverso richieste di amicizia. La piattaforma faciliterà la comunicazione intra-utente, con notifiche per messaggi e la gestione delle amicizie. Gli utenti avranno la libertà di creare e gestire i propri server, con funzionalità per la gestione di canali testuali e multimediali. Sarà possibile inviare messaggi, partecipare a chiamate vocali e video e gestire webcam e microfono all'interno dei canali. Inoltre, i creatori dei server potranno moderarli, rimuovendo membri indesiderati.

[1] https://discord.com

Indice

1. Obiettivi e Requisiti

Il progetto si concentra sulla creazione di una piattaforma per la messaggistica e la comunicazione multimediale ispirata a Discord. Gli obiettivi del progetto includono:

- Sviluppo dell'intera applicazione usando un architettura orientata ai micro-servizi.
- Sviluppo di una piattaforma di chat che permetta di comunicare in tempo reale tra utenti attraverso chat private oppure tramite canali testuali all'interno di server.
- Implementazione di un sistema di comunicazione multimediale che permetta agli utenti di parlare in tempo reale tra loro attraverso canali che supportano audio e video oppure chiamate infra-utenti.
- Creazione di un sistema di notifiche che permetta agli utenti di essere informati quando qualcuno invia messaggi o richieste di amicizia.

Termine	**Definizione**
Server	Raggruppamento di canali
Canale	Luogo dove più utenti possono comunicare. Può essere *testuale* o *multimediale*
Messaggio	Informazioni, in formato testuale, scambiate fra due utenti o all'interno di un canale testuale
Sessione	Comunicazione audio/video fra due utenti o all'interno di un canale multimediale
Direct	Interazione privata tra due utenti

Tabella 1: Glossario dei termini

1.1. Casi d'uso

1.1.1. Utente che accede al servizio

Un utente, che accede al servizio Piperchat, si trova davanti a due scelte:

1. Accedere al servizio autenticandosi, che prevede la necessità di essersi registrati precedentemente.
2. Visualizzare la *Dashboard* che permette il monitoring dello stato dei servizi.

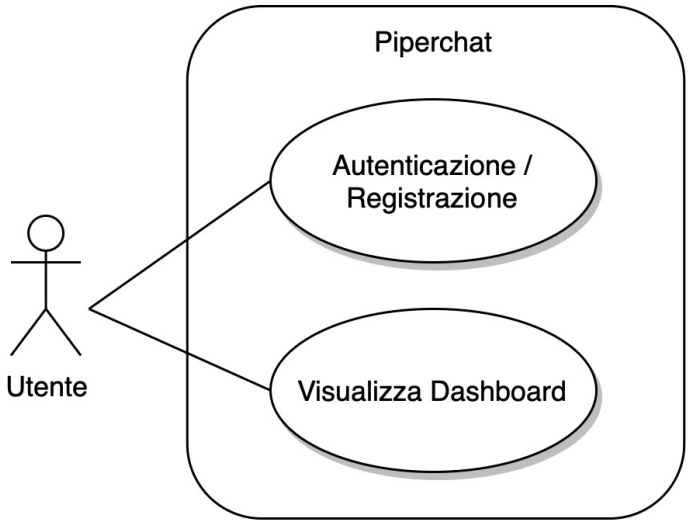


Figura 1: Diagramma dei casi d'uso di un utente non autenticato

1.1.2. Utente autenticato

Dopo aver eseguito l'autenticazione, un utente acquisisce la possibilità di svolgere numerose azione, che riguardano diversi ambiti. Si possono identificare i seguenti scenari:

1. L'utente ha ora possibilità di gestire il proprio profilo ed è abilitato a ricevere le notifiche a lui indirizzate.
2. L'utente può utilizzare la gestione delle richieste di amicizia che prevede di poter a) inviarne ad altri utenti, b) accettare le richieste ricevute c) rifiutare le richieste ricevute.
3. L'utente ha la possibilità di creare server oppure partecipare a server già creati.

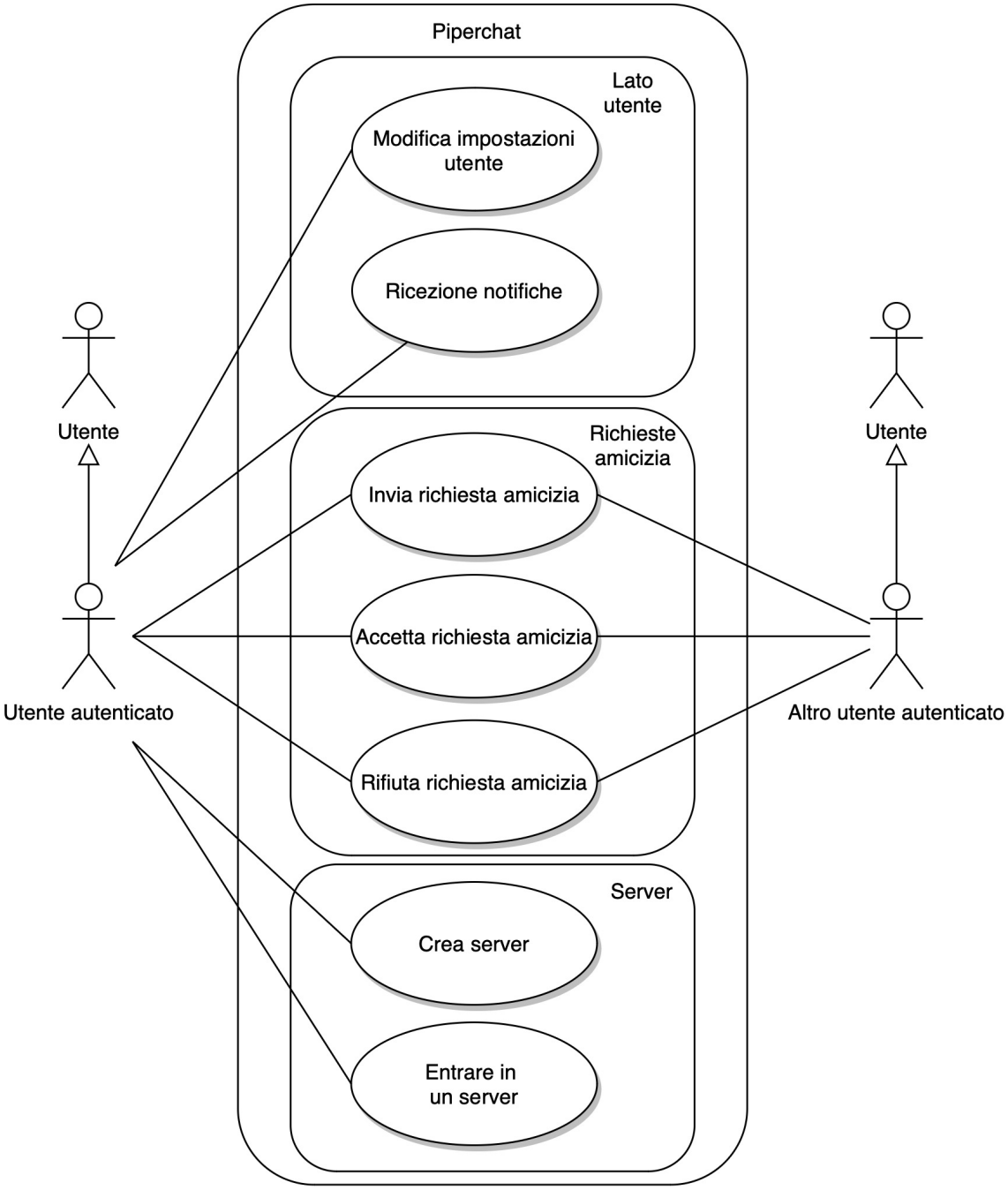


Figura 2: Diagramma dei casi d'uso di un utente autenticato

1.1.3. Utente amministratore di server

Un utente, dopo aver creato un server, ne diventa amministratore. Questo permette di accedere alle funzionalità di gestione per esso, tra le quali si possono evidenziare

le seguenti:

1. L'amministratore può aggiornare le informazioni del server o eliminarlo.
2. L'amministratore può rimuovere un utente dal server.
3. L'amministratore può creare i canali (testuali o multimediali).
4. L'amministratore può aggiornare o rimuovere i canali già creati.

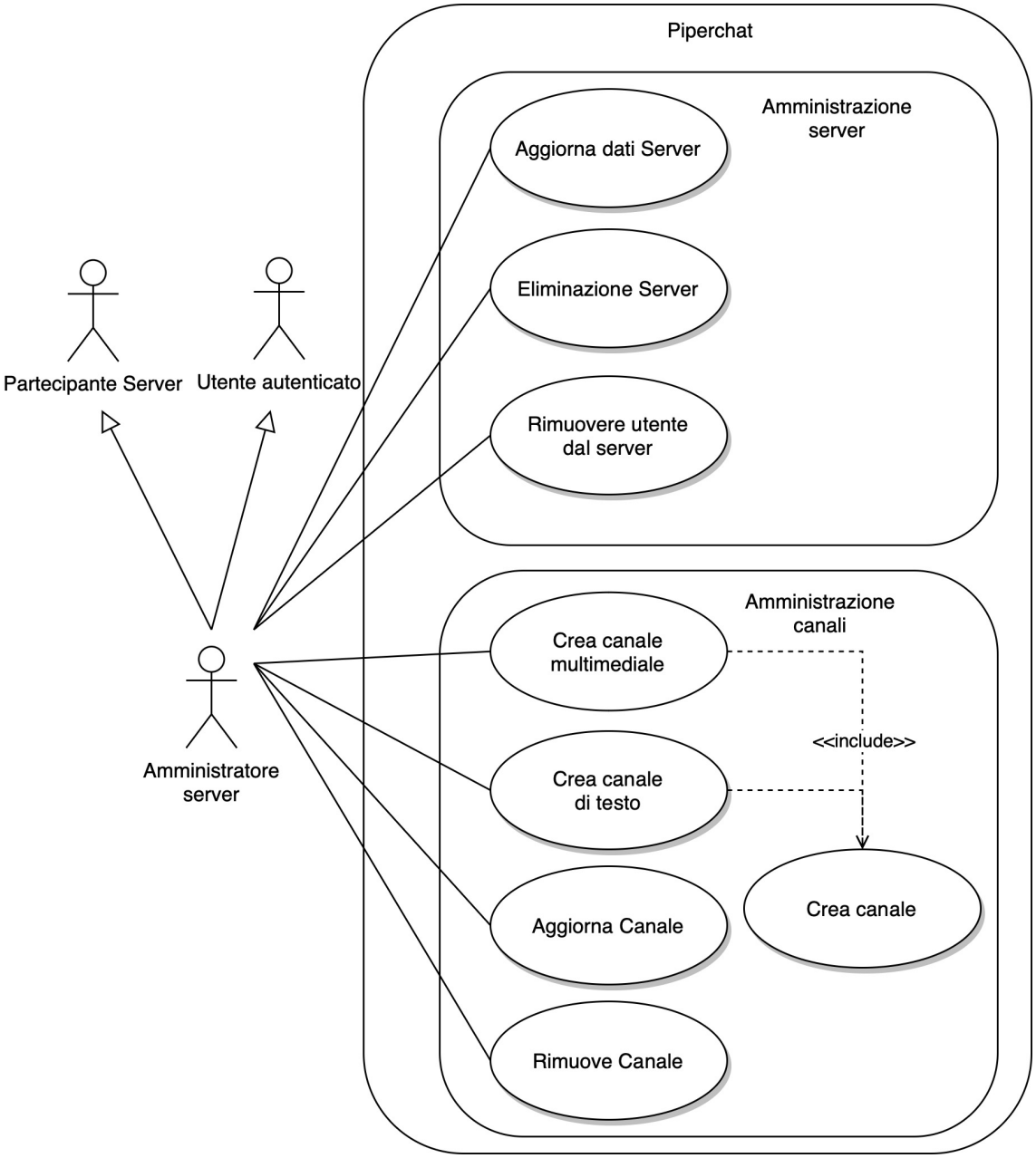


Figura 3: Diagramma dei casi d'uso di un utente amministrazione di un server

1.1.4. Interazione tra utente e amici

Due utenti, dopo aver stretto amicizia, hanno la possibilità di interagire fra loro nei seguenti modi:

1. Inviare messaggi all'interno della chat tra i due utenti.
2. Partecipare alla sessione multimediale.

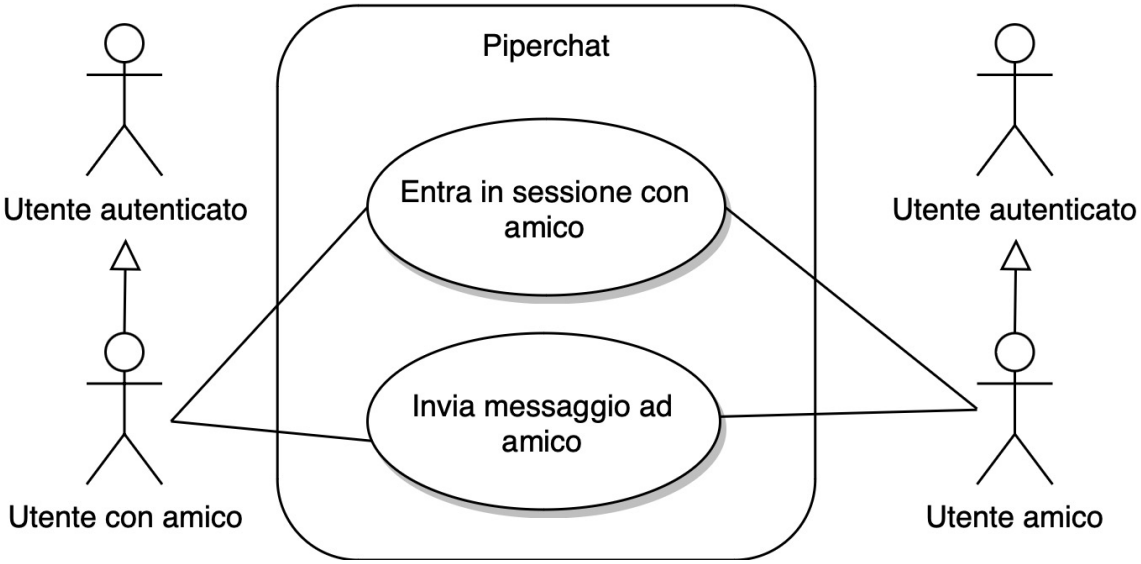


Figura 4: Diagramma dei casi d'uso di un utente che interagisce con un amico

1.1.5. Interazione di un utente partecipante ad un server

Un utente che partecipa ad un server ha la possibilità di interazione attraverso i canali che sono stati creati, nei quali può:

1. Inviare messaggi all'interno dei canali testuali.
2. Partecipare ad un sessione in un canale multimediale.

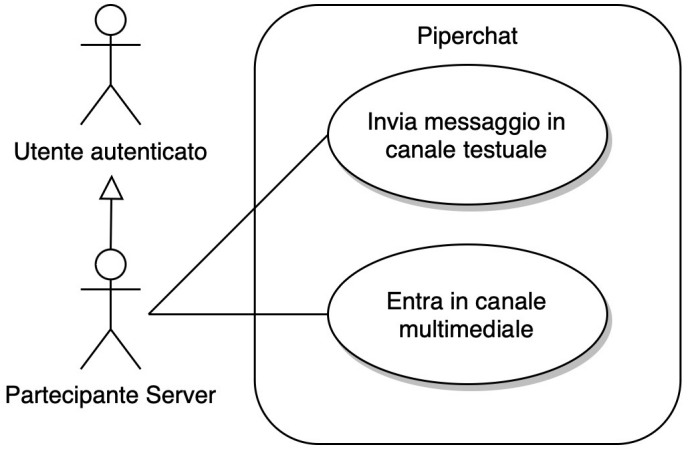


Figura 5: Diagramma dei casi d'uso di un utente che partecipa ad un server

1.1.6. Utente in sessione multimediale

Un utente in sessione multimediale, sia in una privata con un amico, che in un canale, ha la possibilità di gestire microfono e webcam e di uscire dalla sessione stessa.

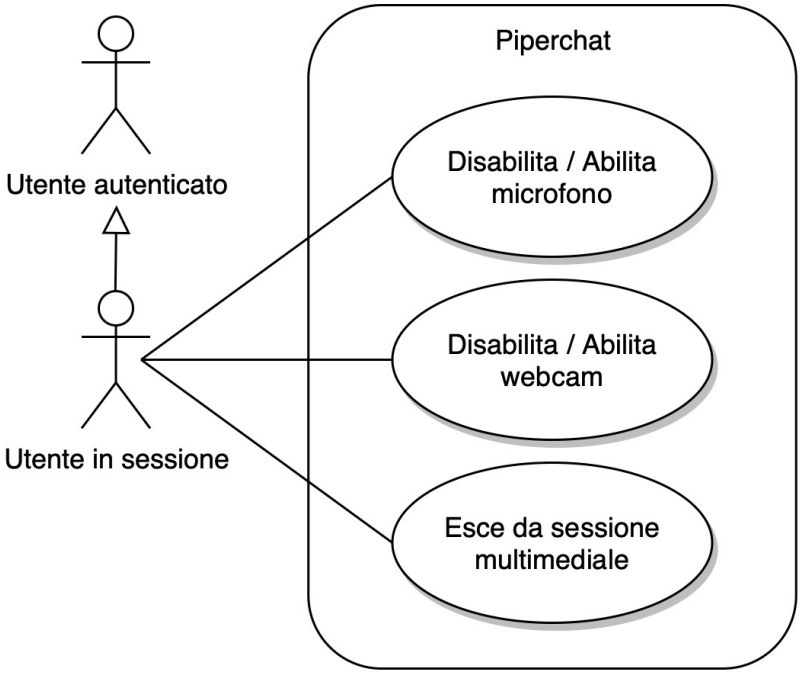


Figura 6: Diagramma dei casi d'uso di un utente che partecipa ad una sessione

1.2. Politica di autovalutazione

Nella valutazione della *qualità* del *software prodotto* per il progetto *PiperChat*, verrà adottato un approccio che tiene in considerazione i seguenti fattori:

- **Funzionalità:** Le funzionalità principali del software, inclusi l'autenticazione degli utenti, la gestione delle amicizie, la messaggistica, la creazione di server e le funzionalità dei canali, devono essere implementate ed efficaci.
- **Affidabilità:** si valuterà la stabilità e l'affidabilità dell'architettura a microservizi, garantendo una comunicazione e un'interazione senza intoppi tra i diversi componenti.
- **Scalabilità:** verrà valutato se l'applicazione avrà la capacità di gestire l'aumentare del carico computazionale legato all'aumentare degli utenti, le interazioni degli stessi, server e canali. Si terrà in considerazione la possibilità di scalabilità orizzontale indipendente del singolo microservizio in modo da consentire una gestione efficiente delle risorse.
- **Sicurezza:** per quanto concerne gli aspetti di sicurezza, renderemo necessaria l'autenticazione nelle richieste, per garantire la riservatezza e l'integrità delle informazioni degli utenti.

Per valutare l'*efficacia* degli esiti del progetto, verranno considerati i seguenti criteri:

- **Decomposizione efficace:** verrà verificata la corretta decomposizione delle funzionalità in microservizi, garantendo che ciascun servizio abbia un ambito ben definito e che le interazioni tra i microservizi siano gestite in modo consono.
- **Dipendenze minime tra Microservizi:** si cercherà di minimizzare le dipendenze tra i microservizi, favorendo l'indipendenza e la modularità per semplificare la gestione e favorire la manutenibilità.
- **Reattività agli Eventi:** verrà assicurata una risposta pronta agli eventi attraverso l'implementazione di meccanismi di *event-driven architecture* tra i microservizi e verso gli utenti finali.
- **Monitoraggio:** verrà implementato un sistema robusto di monitoraggio per i microservizi, permettendo di osservare l'attuale stato dei microservizi in gioco.

2. Analisi dei Requirements

Di seguito vengono formalizzati i requisiti:

2.1. Funzionali

1. **Registrazione e Autenticazione**:
   1.1. Possibilità di registrazione al sistema.
   1.2. Possibilità di login nel sistema.
2. **Sistema di amicizie**:
   2.1. Possibilità di inviare richieste di amicizia ad altri utenti.
   2.2. Possibilità di accettare o rifiutare richieste di amicizia.
   2.3. Sistema di notifiche per la ricezione di nuovo richieste di amicizia.
   2.4. Sistema di gestione dello stato (online) e dell'ultimo accesso degli utenti.
3. **Interazioni con amici**:
   3.1. Gestione messaggistica tra i propri amici.
   3.2. Sistema di notifiche per l'arrivo di nuovi messaggi.
   3.3. Possibilità di entrare in sessione con un amico.
4. **Gestione Server**:
   4.1. Possibilità di creazione di un nuovo server.
   4.2. Possibilità di entrare un server esistente.
   4.3. Possibilità da parte del creatore del server di rimuovere i membri.
5. **Gestione canali**:
   5.1. Possibilità di creazione di canali (testuali o multimediali) da parte del creatore del server.
   5.2. Possibilità di rimozione di canali da parte del creatore del server.

5.3. Sistema di messaggistica per i canali testuali.

5.4. Sistema di notifiche per i nuovi messaggi inviati all'interno di canali testuali.

5.5. Possibilità di accedere alla sessione dei canali multimediali.

6. **Gestione sessione**:

6.1. Possibilità di comunicare attraverso gli altri partecipanti alla stessa sessione.

6.2. Possibilità di accendere e spegnere il microfono e la fotocamera.

7. **Sistema di monitoring dei microservizi**:

7.1. Sistema di monitoraggio dello stato dei servizi.

2.2. Non funzionali

8. **Sicurezza**:

8.1. Autenticazione degli utenti per verificarne l'identità.

8.2. Autorizzazione degli utenti per l'accesso alle risorse in base alle regole stabilite.

8.3. Crittografia per garantire la confidenzialità delle password degli utenti.

9. **Scalabilità**:

9.1. Scalabilità orizzontale per gestire l'aumento del carico operativo.

10. **Manutenibilità**:

10.1. Modularità degli artefatti.

10.2. Codice sorgente ben strutturato e comprensibile.

3. Design

3.1. Dominio

Il dominio di PiperChat ruota attorno ai seguenti concetti chiave: gli *Utenti*, le *Amicizie*, i *Server* e *Canali*.

Un utente, dopo essersi registrato ed autenticato al sistema, si trova di fronte a due possibilità: poter interagire privatamente con un altri utenti, oppure partecipare a dei server.

3.1.1. Interazione tramite Direct

Al fine di permettere l'interazione mediante i *Direct*, due utenti devono, preventivamente, stringere un legame di *amicizia*.

Successivamente ogni utente ha la possibilità di inviare messaggi privati ai propri amici, oppure partecipare ad una sessione.

3.1.2. Interazione tramite server

Un utente può partecipare oppure creare dei *Server*.

Alla creazione di un server, l'utente ne diventa proprietario (o *amministratore*). Questo gli permette di accedere alle varie funzionalità di amministrazione come la modifica delle impostazioni, la rimozione degli altri partecipanti e la creazione di canali.

I *Canali* possono essere:

- **Testuali:** luogo dove gli utenti possono inviare messaggi.
- **Multimediali:** luogo dove gli utenti possono partecipare ad una sessione.

3.1.3. Struttura e relazioni

Di seguito è riportata la struttura del dominio e le interazioni tra le entità precedentemente descritti.

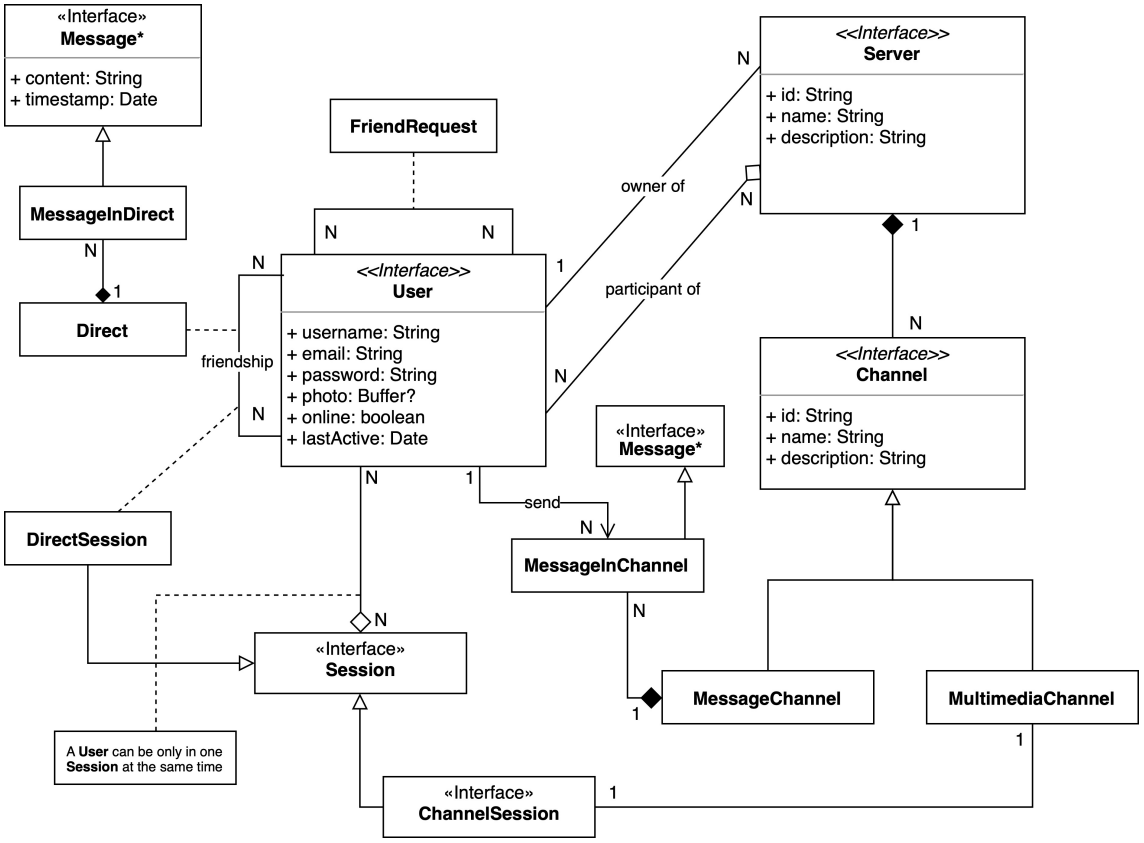


Figura 7: Diagramma delle classi del dominio e le loro relazioni

3.2. Microservizi

Per la realizzzione del sistama si è deciso di adottare un'architettura a *microservizi*, in modo da permette di suddividere la complessità in parti più piccole ad alta coesione, ma accoppiate in modo lasco tra loro. Inoltre, ogni microservizio, se necessario, dispone di un *database*, al quale può accedervi in modo esclusivo.

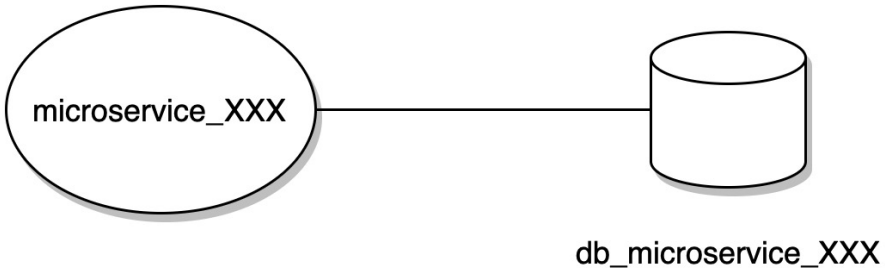


Figura 8: Un esempio di microservizio con il proprio database

A fronte di ciò sono stati identificati i seguenti microservizi:

- **Frontend:** Fornisce l'accesso al sistema servendo la logica del client come *Single Page Application*;
- **Messages:** Gestisce i messaggi degli utenti, sia nelle chat individuali, che per i canali testuali;
- **Monitoring:** Permette il monitoraggio degli altri microservizi;
- **Multimedia:** Gestisce tutto ciò che concerne *WebRTC*, permettendo di istanziare sessioni (e quindi chiamate video-audio) all'interno del sistema.
- **Notifications:** Permette la gestione delle notifiche e lo status degli utenti del sistema (online, ultimo accesso);
- **Piperchat:** Gestisce la struttura di server e dei canali del sistema;
- **Users:** Gestisce l'autenticazione degli utenti al sistema e tutti i dati relativi agli stessi. Inoltre, si occupa di gestire le amicizie fra utenti;

3.2.1. Moduli

Per quanto riguarda la gestione dei moduli dell'applicazione si è scelto di optare per lo sviluppo di un modulo indipendente per ogni servizio.

Tuttavia sono presenti i seguenti moduli comuni:

- **Commons**: contiene le funzionalità/utilities comuni a tutti i servizi.
- **Api**: contiene la definizione delle varie API, con le relative interfacce di richieste e risposte in modo da permettere tipizzazione sia lato backend, che lato frontend in cui sappiamo che richieste inviare e che risposte aspettarci.
- **Messages-Api**: contiene la definizione di tutte le tipologie di eventi che vengono inviati al broker dai vari servizi. Rappresenta l'insieme degli eventi interni al sistema.

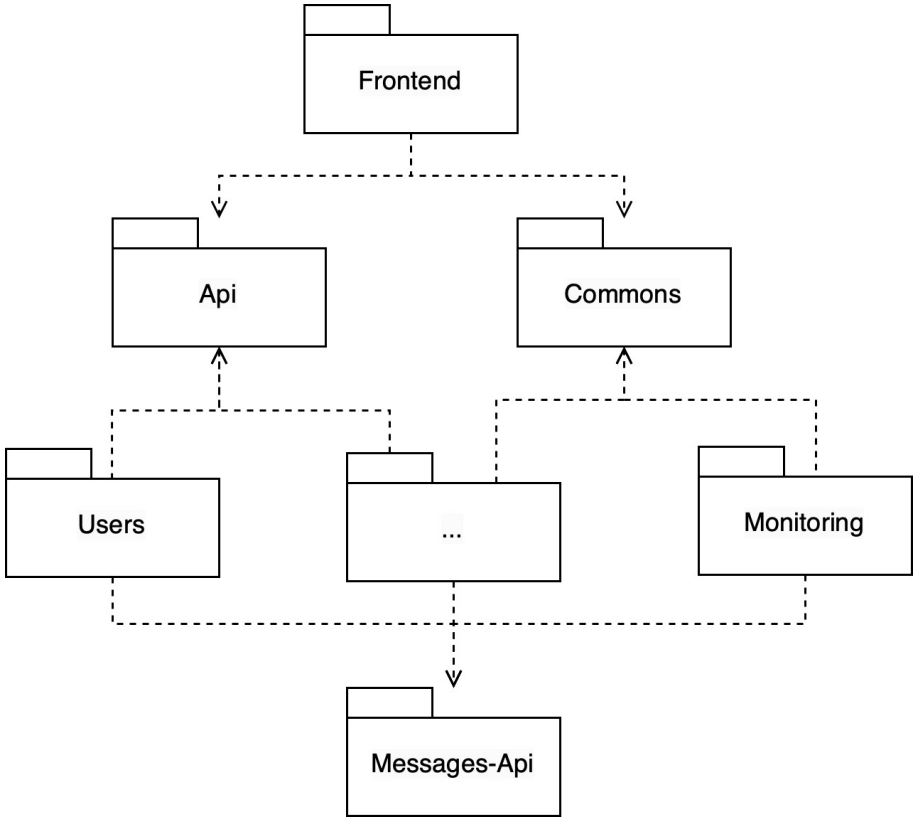


Figura 9: Diagrammi dei package Piperchat

3.3. Comportamento

Di seguito, vengono descritti nel dettaglio i comportamenti dei vari microservizi presenti nel sistema.

3.3.1. Users Service

Microservizio responsabile della gestione degli utenti e tutte le operazioni relative a quest'ultimi, fra cui:

- Creazione degli utenti nel sistema (Registrazione)
- Autenticazione degli utenti nel sistema (Login)
- Modifiche al profilo dei singoli utenti del sistema
- Gestione delle amicizie fra utenti

3.3.2. Piperchat Service

Microservizio relativo ai server e ai canali, gestisce tutte le operazioni quali:

- Operazioni CRUD relative al concetto di Server
- Operazioni CRUD relative al concetto di Canali all'interno di Server

3.3.3. Messages Service

Microservizio relativo alla messaggistica. Si occupa della gestione delle comunicazioni testuali tra gli utenti all'interno della piattaforma PiperChat, sia per chat dirette tra utenti che per i canali testuali.

- **Messaggi Diretti:** il microservizio gestisce l'invio e la ricezione di messaggi diretti tra due utenti. Questa funzionalità è utilizzata per le conversazioni private.
- **Canali di Chat:** permette agli utenti di utilizzare le chat all'interno dei canali testuali dei server.
- **Gestione dei Messaggi:** il microservizio gestisce l'archiviazione e il recupero dei messaggi inviati, consentendo agli utenti di accedere alla cronologia dei messaggi in un canale specifico o in una conversazione diretta.

3.3.4. Multimedia Service

Il microservizio si occupa della gestione delle chiamate multimediali tramite l'utilizzo del concetto di sessione multimediale, sia per le conversazioni dirette tra due utenti, che per i canali.

3.3.5. Notifications Service

Il microservizio si occupa di gestire la comunicazione real-time con i client. Esso si occupa sia di inviare al client tutte le notifiche su eventi che lo riguardano e tiene inoltre traccia degli utenti connessi, andando a rendere disponibili tali informazioni tramite API per ottenere lo stato di un determinato utente.

3.3.6. Monitoring Service

Il microservizio si occupa del monitoraggio dello stato operativo dei servizi, eseguendo regolarmente verifiche per stabilire se si trovino in uno stato online o offline.

3.4. Interazione

Al fine di garantire sia una comunicazione interna fra i microservizi, che una comunicazione verso l'esterno, sono stati identificati e realizzati i seguenti componenti:

- **API Gateway:** componente che permette la comunicazione tra i client esterni al sistema, con i microservizi. Si occupa di redirezionare le richieste agli appositi servizi.
- **Broker:** componente che permette la comunicazione interna al sistema, fra i microservizi stessi.

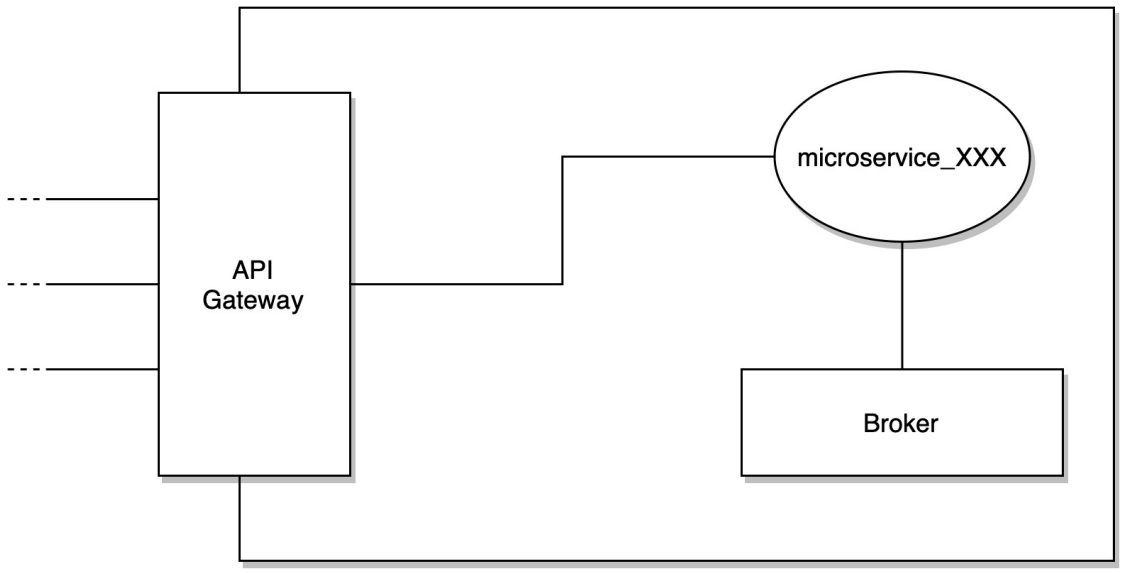
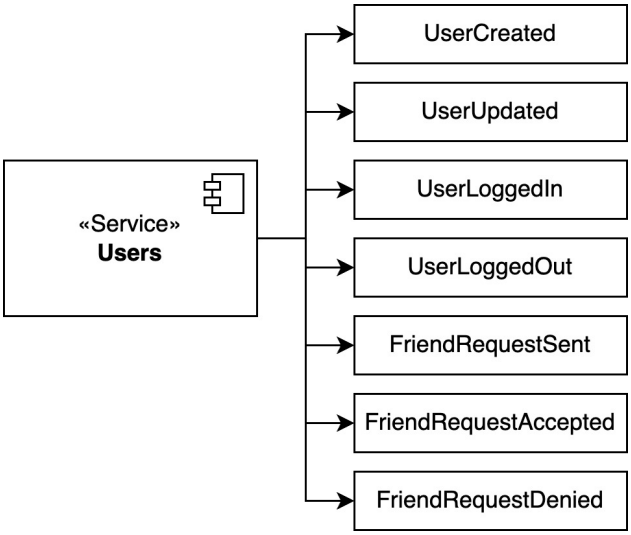


Figura 10: Un microservizio collegato al Broker e all'API Gateway

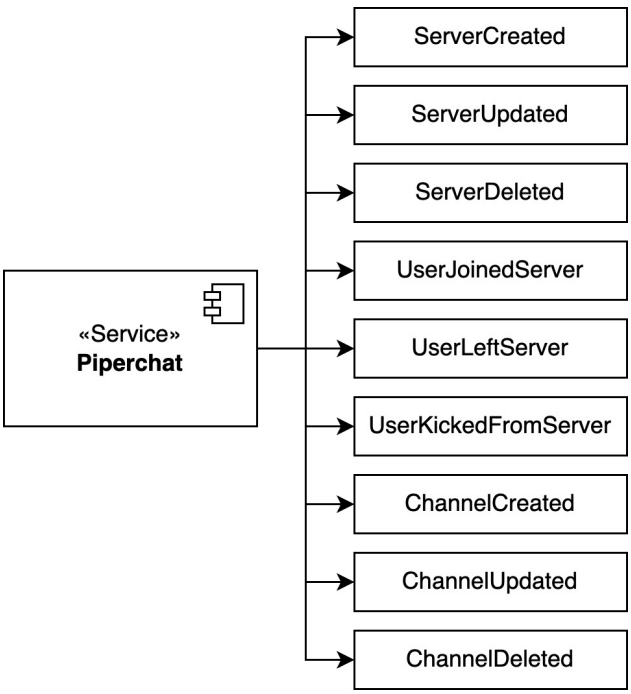
3.4.1. Eventi

Di seguito sono riportate le interazioni che hanno i microservizi tra loro per quanto riguarda gli eventi che ricevono ed inviano:

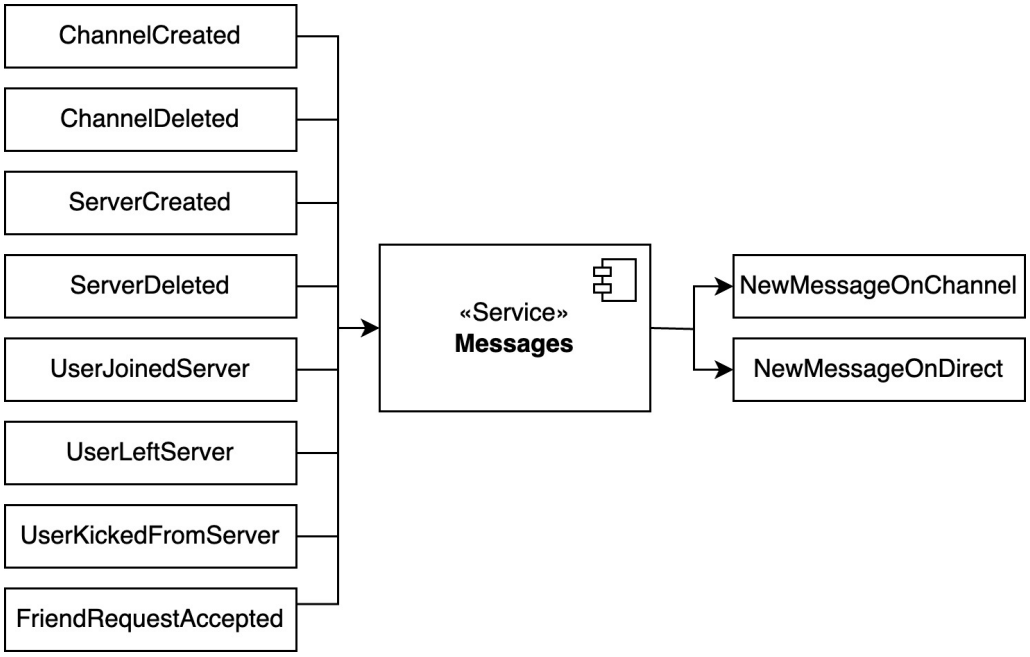
Servizio Utenti Per quanto riguarda il servizio degli utenti, esso non ascolta nessuna tipologia di evento ma produce tutti gli eventi relative alla creazione/modifica degli utenti e delle richieste di amicizia in modo che gli altri servizi possono ricostruire lo stato degli utenti e delle amicizie tra di loro.



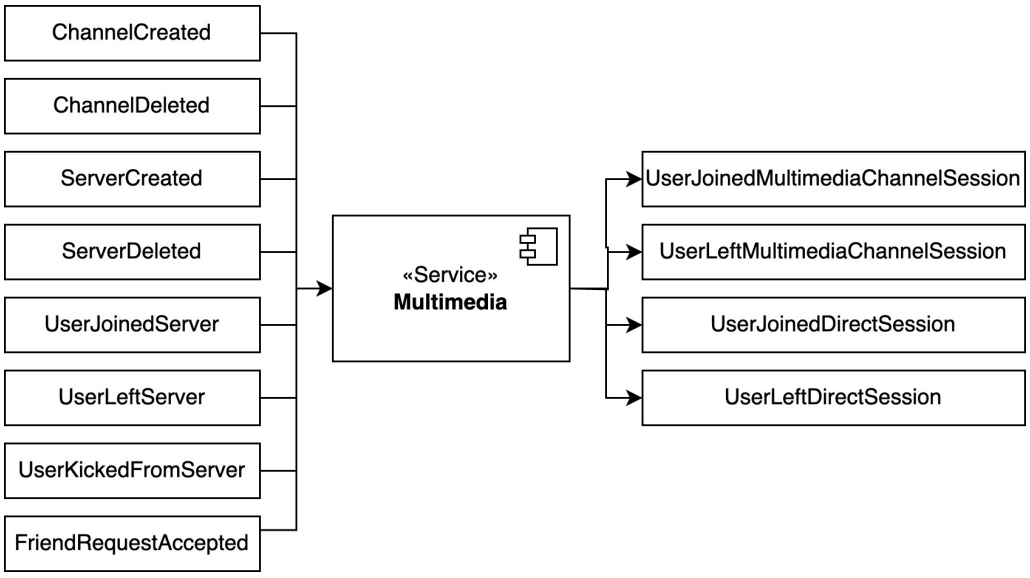
Servizio Piperchat Per quanto riguarda il servizio Piperchat anch'esso non ascolta nulla e produce tutti gli eventi relativi ai server/canali e alle partecipazioni/abbandoni dei server da parte degli utenti. In questo modo gli altri servizi possono rimanere sincronizzati su tali informazioni.



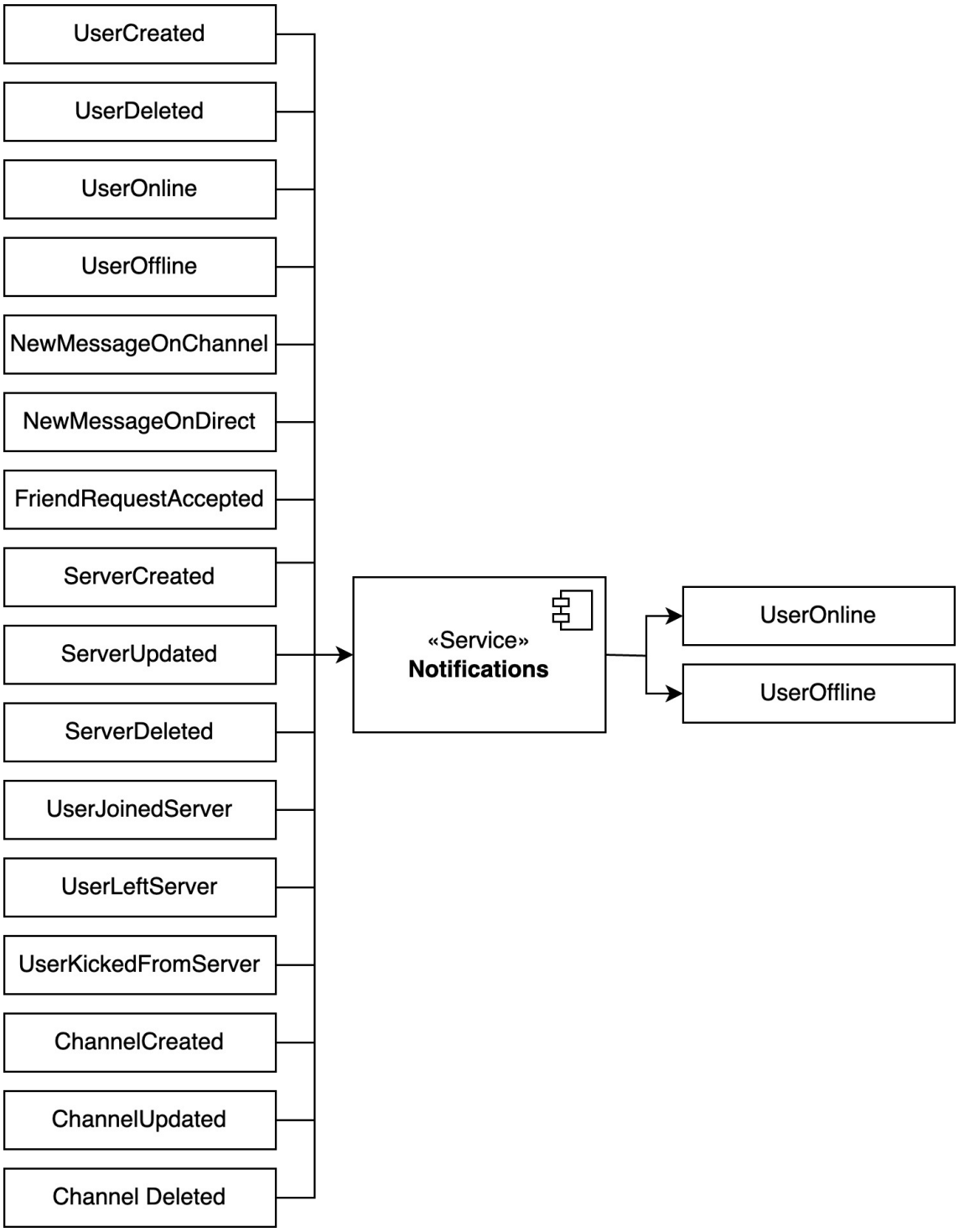
Servizio Messaggi Il servizio messaggi rimane in ascolto per tutti gli eventi che riguardano i canali e le amicizie in modo da sapere se l'invio di un messaggio è possibile. Inoltre si occupa della produzione degli eventi riguardanti l'invio di nuovi messaggi.



Servizio Multimedia Il servizio che si occupa della gestione delle sessioni multimediali rimane in ascolto degli eventi riguardanti i server e le amicizie anch'esso per capire quando una sessione multimediale tra due utenti o in un server possa essere effettuata. Si occupa anche della pubblicazione degli eventi riguardanti i join/left delle sessioni da parte degli utenti.



Servizio Notifiche Il servizio di notifiche ascolta la maggior parte degli eventi che propaga agli utenti interessati in modo da rendere l'interfaccia e l'interazione con il servizio reattivi. Tiene inoltre traccia degli utenti che sono a lui connessi per stabilire il loro status (online/offline/ultimo accesso).



3.5. Recap dell'architettura proposta

Un utente, per accedere al servizio sfrutta l'*API Gateway*. In questo modo è possibile nascondere l'implementazione retrostante.

Di seguito è riportato lo schema architetturale del sistema.

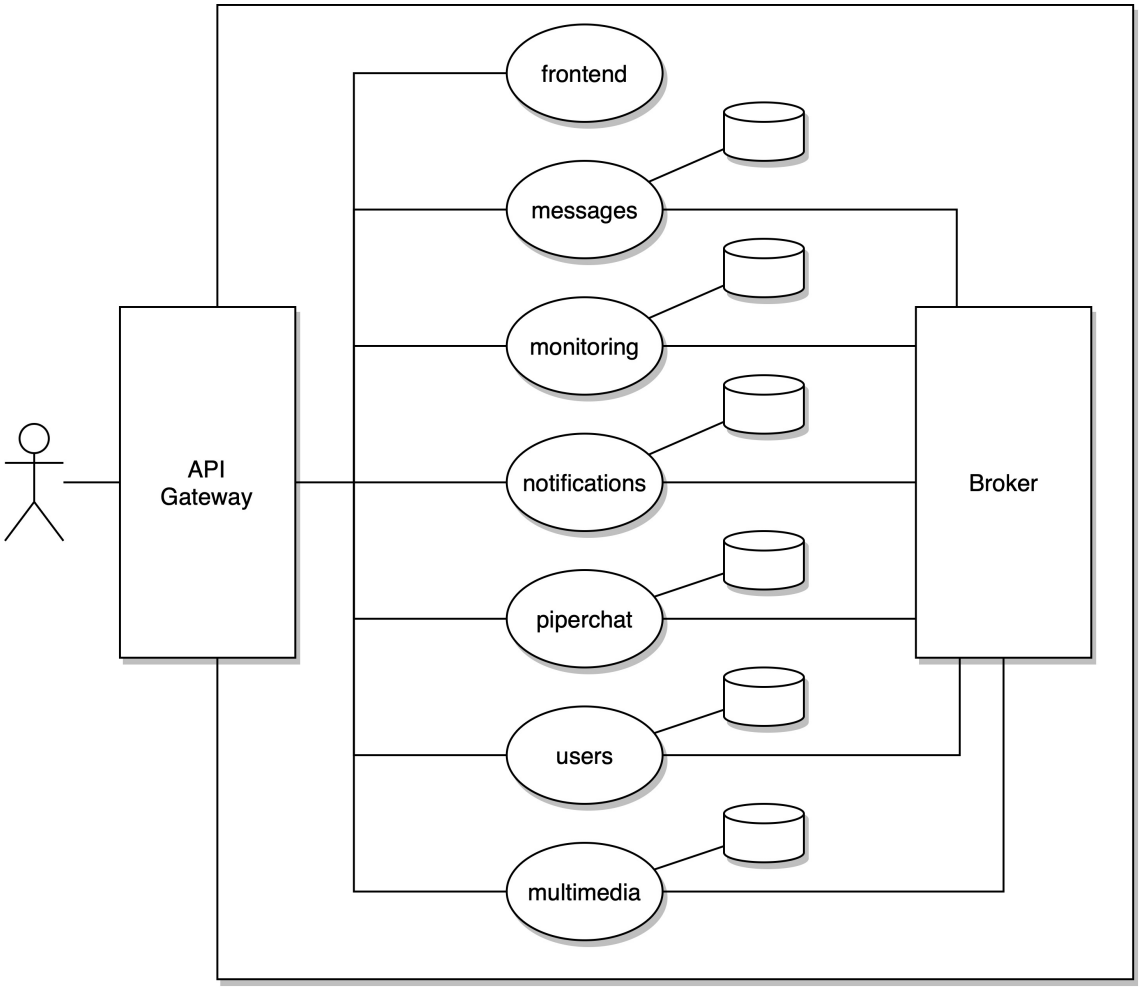


Figura 11: Architettura di Piperchat

4. Dettagli implementativi

4.1. Documentazione

4.1.1. Swagger

Swagger è un insieme di strumenti open source per progettare, creare, documentare e consumare servizi web RESTful, attraverso la specifica *Open API*. L'obiettivo principale di Swagger è semplificare e standardizzare il processo di sviluppo e integrazione di API, fornendo una documentazione interattiva e uno strumento per testare direttamente le API.

Le caratteristiche chiave di Swagger includono:

- Definizione della API: Swagger utilizza il formato YAML o JSON per definire la struttura e i dettagli di un'API RESTful, specificando risorse, operazioni, parametri, tipi di dati, e altro ancora.
- Interfaccia Utente Interattiva: Genera automaticamente un'interfaccia utente interattiva (Swagger UI) basata sulla definizione dell'API. Questa interfaccia consente agli sviluppatori di esplorare e testare le API direttamente dal browser.
- Standardizzazione e Conformità: Swagger promuove la standardizzazione nelle API RESTful, migliorando la coerenza e la comprensione tra sviluppatori e team di sviluppo

Utilizzo Con Swagger, è stato possibile creare documentazione dettagliata delle API, specificando dettagli come gli endpoint, i parametri, i tipi di dati, le risposte possibili e persino esempi di richieste e risposte.

Inizialmente, lo strumento è stato utilizzato per realizzare il design del sistema, permettendo di documentare ciò che si sarebbe dovuto implementare. I cicli iterativi di raffinamento hanno permesso di convergere all'attuale stato dell'*API*.

La documentazione delle API del progetto è consultabile sulle Github Pages: `https://zucchero-sintattico.github.io/piperchat/api/rest/`

4.1.2. AsyncApi

AsyncAPI è uno standard di specifica per la progettazione di API asincrone. Simile a come OpenAPI è utilizzato per definire e documentare API sincrone, AsyncAPI è progettato specificamente per gestire le comunicazioni asincrone, come quelle basate su messaggi o eventi.

Le principali caratteristiche di AsyncAPI includono:

- Comunicazioni Asincrone: AsyncAPI è progettato per modellare API che coinvolgono comunicazioni asincrone, dove la richiesta e la risposta non sono sincronizzate nel tempo.
- Documentazione Dettagliata: Come OpenAPI per API sincrone, AsyncAPI fornisce una specifica dettagliata per documentare aspetti come endpoint, messaggi, schemi dei dati, protocolli di trasporto e altri dettagli relativi alla comunicazione asincrona.
- Supporto per Protocolli Comuni: AsyncAPI supporta una varietà di protocolli comuni per la comunicazione asincrona, come MQTT, AMQP e WebSocket. Ciò consente una flessibilità nella progettazione delle API a seconda delle esigenze specifiche del progetto.

Utilizzo AsyncAPI è stato utilizzato come strumento di supporto e di documentazione di tutte le tipologie di messaggi rappresentanti gli eventi che vengono scambiati tramite il broker all'interno dell'architettura a microservizi.

La documentazione delle API per i messaggi *infra-servizi* è consultabile sulle Github Pages: `https://zucchero-sintattico.github.io/piperchat/api/infra-service/`

Inoltre, la tecnologia è stata sfruttata anche per la documentazione dei messaggi *inviati ai client* come notifiche di eventi. Tale documentazione è disponibile al seguente link: `https://zucchero-sintattico.github.io/piperchat/api/notification/`

4.2. View

Per la realizzazione del software in oggetto, abbiamo fatto ampio uso del framework *Vue.js*[2], un potente e flessibile framework JavaScript per la costruzione di interfacce utente moderne e reattive. Vue.js si è rivelato una scelta eccellente per la nostra applicazione, offrendo una struttura chiara e modulare che ha semplificato lo sviluppo e la manutenzione del codice. La sua capacità di gestire in modo efficiente la visualizzazione dinamica dei dati e la reattività dell'interfaccia utente ha contribuito in modo significativo a garantire un'esperienza utente fluida e coinvolgente.

4.2.1. Pinia

Pinia JS[3] è una libreria di gestione dello stato progettata per applicazioni Vue.js. Essa fornisce un'architettura di gestione dello stato centralizzata e reattiva, offrendo uno store centralizzato per memorizzare e gestire lo stato dell'applicazione. Pinia si basa sui concetti principali di Vue.js, come la reattività e la gestione delle modifiche dello stato in modo efficiente. Questo strumento consente agli sviluppatori di scrivere codice pulito e manutenibile, facilitando la gestione dello stato dell'applicazione Vue.js.

4.2.2. Quasar

Quasar[4] è un framework open-source basato su Vue.js, progettato per semplificare lo sviluppo di applicazioni web e mobile con un unico codice sorgente. È noto per la sua flessibilità e la capacità di generare applicazioni per diverse piattaforme. Le caratteristiche principali di Quasar Framework includono:

1. **Componenti Vue.js predefiniti**: Quasar offre una vasta libreria di componenti Vue.js personalizzati e ricchi di funzionalità, che semplificano la creazione di interfacce utente sofisticate.

2. **Responsive Design**: Le applicazioni Quasar possono essere facilmente rese responsive per adattarsi a diversi dispositivi e dimensioni dello schermo.

[2] `https://vuejs.org`
[3] `https://pinia.vuejs.org`
[4] `https://quasar.dev`

3. **Material Design**: Quasar aderisce a Material Design di Google, offrendo un aspetto moderno e uniforme per le applicazioni.

4.3. Api Gateway

4.3.1. Traefik

Traefik[5] è un moderno *reverse proxy* e *load balancer* progettato per gestire il traffico web in ambienti complessi e distribuiti.

Utilizzo Nel contesto del nostro progetto, l'utilizzo di Traefik è stato utilizzato per realizzare un *API Gateway*, instradando e distribuendo il traffico tra il frontend e i diversi microservizi del backend. All'interno di ciascun file *Docker Compose* relativo ai singoli servizi, sono state specificate le rotte accettate dal microservizio attraverso l'aggiunta di una *label*[6]. Questo approccio ci ha permesso di configurare facilmente e in modo dettagliato le direttive per il routing del traffico verso ciascun servizio, consentendo a Traefik di instradare le richieste in base alle specifiche esigenze di ciascun microservizio.

Di seguito è riportato un esempio della definizione delle rotte.

```
# Compose file

service:
  users-service:
    ...
    labels:
      - |
        traefik.http.routers.users-service.rule=
        (Method(‘GET‘) && Path(‘/friends‘)) ||
        (Method(‘GET‘) && Path(‘/whoami‘)) ||
        (Method(‘GET‘, ‘POST‘) && Path(‘/friends/requests‘)) ||
        ...
```

[5] https://traefik.io/traefik/

[6] https://doc.traefik.io/traefik/providers/docker/#routing-configuration-with-labels

4.4. Gestione comunicazione persistente

Per quanto riguarda la gestione delle comunicazioni persistenti tra backend e client è stata utilizzata la libreria *Socket.io*, impiegata sia nel servizio di notifiche che nel servizio multimediale per propagare i messaggi nel protocollo di join di una sessione multimediale.

4.4.1. Socket.io

Socket.IO è una libreria JavaScript che fornisce una comunicazione bidirezionale in tempo reale tra il server e il client in applicazioni web. È spesso utilizzata per implementare funzionalità di chat, giochi multiplayer, aggiornamenti in tempo reale e altre applicazioni che richiedono una comunicazione immediata tra il server e il browser. Le principali caratteristiche di Socket.IO per cui è stato scelto come framework sono:

- **Comunicazione in tempo reale**: una comunicazione bidirezionale in tempo reale, consentendo agli utenti di ricevere aggiornamenti istantanei dal server.
- **Riconnessione automatica**: è supportata la riconnessione automatica in caso di perdita di connessione, garantendo che gli utenti rimangano connessi.

4.4.2. Notifications Service: notifiche e gestione status

Un utente, quando si connette al sistema, deve essere considerato online e deve essere abilitato alla ricezione delle notifiche.

Date queste premesse, viene instaurata, tra il server e il client autenticato, una connessione tramite Socket.IO. Si è deciso di collassare, all'interno di questa socket, le due responsabilità:

1. Permettere al server di inviare notifiche ai client.
2. Finché la socket è aperta, il client viene considerato online.

Considerazioni su scalabilità e stato persistente Dal momento che il micro-servizio mantiene uno stato, la scalabilità orizzontale potrebbe non essere ovvia.

Di seguito vengono analizzate le considerazioni effettuate al fine di permettere la scalabilità orizzontale, senza incorrere in inconsistenze.

Utente stabilisce la connessione Quando un utente fa richiesta verso il servizio di notifica, instaura una socket verso una specifica replica. Questa replica sarà colei che manterrà la connessione persistente verso l'utente. Inoltre, essa si occupa di aggiornare lo status (online/offline) nel database condiviso con le altre repliche.

Nuovo evento per un utente Quando un nuovo evento viene generato nel sistema ed un utente, deve essere notificato. Il broker invierà in broadcast l'evento a tutte le copie del servizio, ma sarà solo la replica che mantiene la connessione persistente ad inviare al client l'opportuna notifica.

Richiesta dello status di un utente Quando un utente qualsiasi richieste lo stato di un altro utente, qualsiasi replica può assolvere a questa richiesta dal momento che lo status è salvato nel database condiviso.

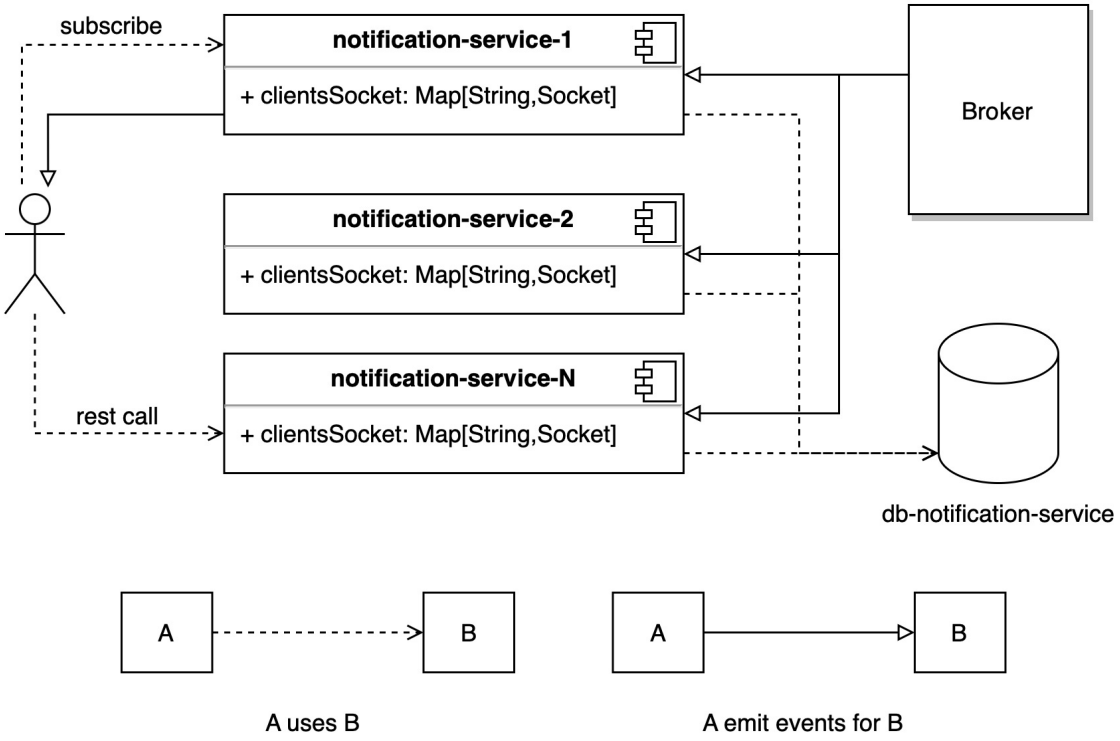


Figura 12: Un utente crea una connessione persistente

4.4.3. Socket.IO vs Server Sent Event

Durante le fasi iniziali del progetto, al fine di inviare le notifiche dal microservizio verso il client, era stata adottato *Server-Sent Events*[7], una tecnologia di comunicazione monodirezionale, che permette di ricevere dati da un server che li invia.

Questo approccio è stato successivamente sostituito da Socket.IO perché il server non può gestire in modo responsivo la chiusura di una connessione da parte del client.

4.5. WebRTC

WebRTC[8], acronimo di "Web Real-Time Communication", è una tecnologia open-source che consente la comunicazione audio e video in tempo reale direttamente tra browser web senza richiedere plugin o software aggiuntivi. È utilizzato per creare applicazioni di videoconferenza, chat video, streaming multimediale e altro.

Le principali caratteristiche di WebRTC includono:

1. **Comunicazione peer-to-peer**: WebRTC consente ai browser di comunicare direttamente tra loro, evitando la necessità di server intermediari per la trasmissione di dati in tempo reale.
2. **Supporto per audio e video**: WebRTC supporta la comunicazione audio e video in tempo reale, consentendo agli utenti di interagire tramite chat video o conferenze online.
3. **Accesso ai dispositivi**: WebRTC consente l'accesso ai dispositivi hardware come telecamere e microfoni per abilitare la cattura audio e video.

4.5.1. Funzionamento

Nel contesto di WebRTC, ci sono tre concetti chiave: **offer**, **answer**, e **ICE candidates**.

[7] `https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events`
[8] `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API`

- **Offer**: L'offerta è il punto di partenza per l'inizializzazione di una connessione WebRTC.

  Un peer (la parte che inizia la connessione) crea un'offerta che specifica le sue preferenze per la sessione di comunicazione, inclusi i codec supportati, i parametri di sicurezza, etc. L'offerta è creata utilizzando l'API *createOffer()*.

- **Answer**: L'answer è generata dal peer destinatario in risposta all'offerta ricevuta. Il peer destinatario, dopo aver ricevuto l'offerta, crea una risposta che riflette le sue preferenze per la comunicazione. L'answer è creata utilizzando l'API *createAnswer()*.

- **ICE Candidates**: ICE (Interactive Connectivity Establishment) è un protocollo utilizzato per stabilire una connessione in presenza di reti complesse, come dietro firewall o NAT. I candidati ICE sono gli indirizzi IP e le porte su cui un peer può essere contattato. Durante il processo di offerta e risposta, i peer scambiano i loro candidati ICE utilizzando l'API *onIceCandidate()*. Il processo di raccolta dei candidati ICE è noto come "ICE gathering".

In sintesi, il flusso tipico di inizializzazione di una connessione WebRTC coinvolge la creazione di un'offerta da parte del peer iniziatore, la trasmissione di questa offerta al peer destinatario, la creazione di una risposta da parte del peer destinatario e, infine, lo scambio di candidati ICE per stabilire una connessione diretta tra i due peer. Questo processo è fondamentale per consentire la comunicazione bidirezionale in tempo reale tra browser senza passare attraverso un server intermedio.

4.5.2. Sessione multimediale

La gestione delle sessioni per le videochiamate WebRTC si basa su un modello che prevede l'utilizzo di sessioni individuali, ciascuna identificata da un unico ID e caratterizzata da un insieme di utenti consentiti ("allowed users") e dagli attuali partecipanti durante la chiamata in corso. Ogni amicizia tra utenti è associata a una propria sessione, e lo stesso vale per i canali multimediali, i quali fanno riferimento a una sessione per agevolare le chiamate multimediali.

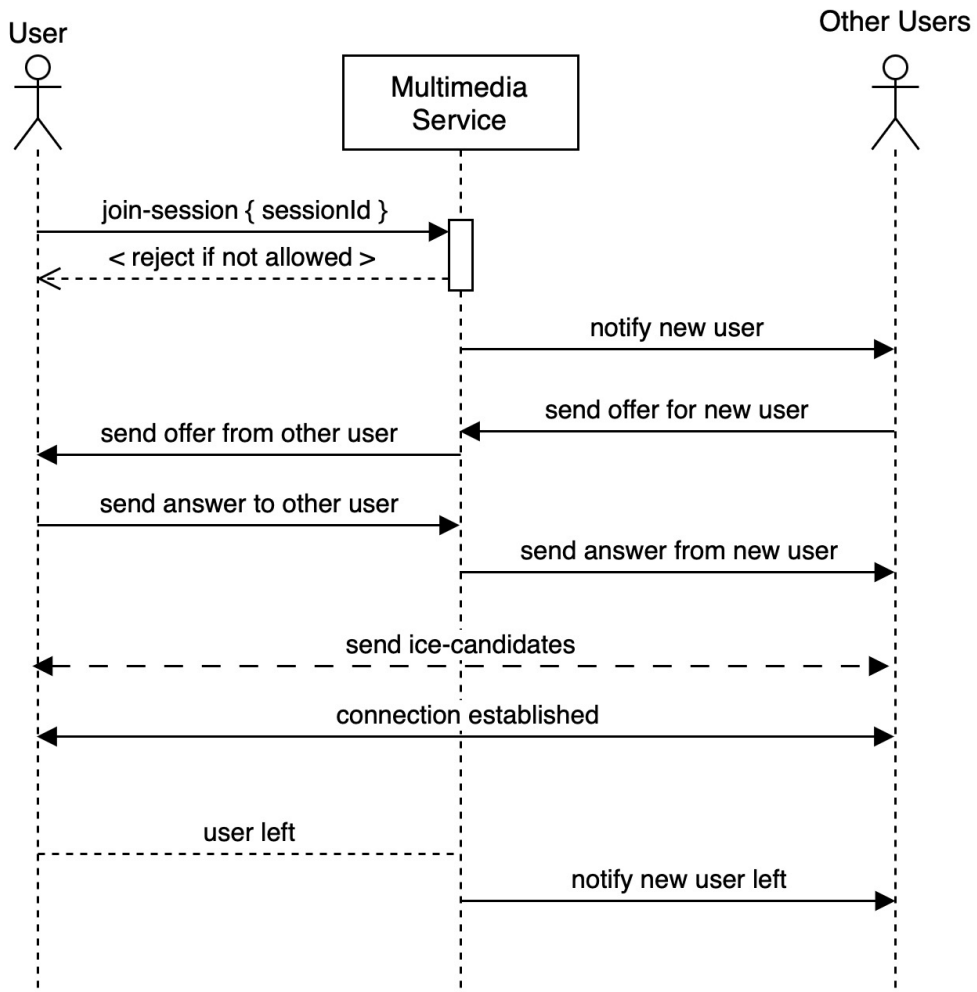
Le sessioni vengono distintamente gestite in base al contesto. Nel caso delle amicizie, l'insieme di utenti consentiti è limitato esclusivamente ai due partecipanti dell'amicizia, garantendo una comunicazione privata tra i diretti interessati. Invece,

nei canali multimediali, l'insieme di utenti consentiti comprende tutti i partecipanti del server, consentendo così chiamate multimediali che coinvolgono più membri contemporaneamente.

Questo approccio alla gestione delle sessioni offre un controllo granulare sull'accesso alle videochiamate, garantendo che la comunicazione sia personalizzata in base al contesto delle relazioni tra gli utenti. Inoltre, consente una scalabilità efficiente per le chiamate multimediali su larga scala, consentendo a tutti i partecipanti del server di partecipare alle conversazioni senza compromettere la sicurezza o la privacy delle comunicazioni one-to-one.

4.5.3. Protocollo Signaling

Di seguito il protocollo utilizzato per la procedura di signaling webrtc che permette lo scambio delle informazioni necessarie all instauramento delle connessioni p2p.



4.5.4. Problema P2P e TURN

Uno dei problemi principali che WebRTC deve affrontare è la presenza di dispositivi di rete con NAT (Network Address Translation) non compatibili, che possono complicare la creazione di connessioni dirette tra i partecipanti.

Il NAT consente a più dispositivi di condividere un singolo indirizzo IP pubblico, fornendo una forma di sicurezza e limitando la quantità di traffico Internet diretto verso dispositivi locali. Tuttavia, questo può creare problemi quando si tenta di stabilire connessioni dirette tramite WebRTC, in quanto alcuni NAT possono impedire il passaggio dei pacchetti dati necessari.

Per superare questo problema, WebRTC utilizza un concetto chiamato *Traversal Using Relays around NAT (TURN)*. Un server TURN agisce come intermediario tra i partecipanti, aiutando a instradare i dati quando le connessioni dirette non sono possibili. Quando due peer non possono stabilire una connessione diretta a causa di NAT non compatibili, i dati vengono instradati attraverso il server TURN, consentendo comunque la comunicazione in tempo reale.

Per ovviare a questa problematica è stato quindi aggiunto al deploy anche un servizio adibito a funzionare da TURN in caso non sia possibile la connessione p2p.

4.6. Autenticazione degli utenti

Per gestire l'autenticazione degli utenti all'interno del nostro sistema, è stata adottata la tecnologia *JSON Web Token (JWT)*.

4.6.1. JWT

JSON Web Token (JWT) è uno standard aperto (RFC 7519) che definisce un modo compatto per rappresentare informazioni tra due parti. Queste informazioni possono essere verificate e fidate, poiché sono firmate digitalmente.

Struttura dei JWT Un JWT è costituito da tre parti separate da punti (.), che sono:

- **Header:** Contiene il tipo di token (che è JWT) e l'algoritmo di firma utilizzato, ad esempio, HMAC SHA256 o RSA.
- **Payload:** Contiene le dichiarazioni, chiamate anche *claim*, che rappresentano l'entità (solitamente l'utente) e le informazioni aggiuntive.
- **Signature:** Per ottenere la firma, si prende l'header codificato, il payload codificato, una chiave segreta, e si applica un algoritmo specifico. Questa firma viene aggiunta al token.

Funzionamento Quando un utente si autentica, riceve un JWT che può includere informazioni come l'identità e i diritti di accesso. Da quel momento in poi, il client invia il JWT con ogni richiesta successiva al server, che può verificare la firma del token per assicurarsi che sia valido. In questo modo, il server può fidarsi delle informazioni contenute nel JWT senza doverle verificare ad ogni richiesta.

Utilizzo Nel contesto di Piperchat, quando un utente si autentica con successo, il microservizio `Users` genera un JWT contenente nel payload le informazioni relative all'identità dell'utente. Questo JWT viene restituito al client, che lo utilizzerà nelle successive richieste per dimostrare la propria autenticità.

Inoltre, viene memorizzato nel database un token di refresh, che consente di ottenere nuovi token di accesso senza dover effettuare nuovamente l'accesso.

Quando il token di accesso scade, il client può richiedere un nuovo token di accesso al server utilizzando il token di refresh. Questo processo aiuta a mantenere un'esperienza utente fluida, riducendo al contempo il rischio di accessi non autorizzati.

4.7. Definizione delle API

Per quanto riguarda la gestione delle api dei vari microservizi si è optato per avere una struttura che rappresentasse ogni endpoint, incapsulando sia i dati richiesti sia le possibili risposte.

A supporto di ciò è stato creato un modulo **api** che incapsula tutti gli endpoint del backend con le relativi informazioni.

Tale modulo viene sfruttato sia dal backend, per avere un typing migliore e un controllo di validazione dei parametri richiesti, che dal frontend per sapere già quali sono codice di risposta e tipologie di risposta per un determinato endpoint.

Listing 1: Definizione API

```
export module LoginApi {

  export module Request {
    export type Body = {
      username: string
      password: string
    }
    export const Schema: RequestSchema = {
      Body: {
        username: 'string',
        password: 'string',
      },
    }
  }

  export module Responses {
    export class Success extends Response {
      statusCode = 200
      message = 'Logged in' as const
      jwt: string
    }
  }

  export module Errors {
    export class UsernameOrPasswordIncorrect extends
       ErrorResponse {
      statusCode = 401
      error = 'Username or password incorrect' as const
    }
  }
}
```

4.8. Definizione degli Endpoint

Una volta costruita la struttura rappresentante le singole API è stata creata l'utility **Route**, che a partire da un api permette di implementare l'endpoint aggiungendo il supporto automatico alla validazione dei dati in modo che se i dati in ingresso non dovessero essere corretti, l'handler non venga notificato e venga restituito un messaggio di errore *Bad Request*. Inoltre permette e di dichiarare come reagire per ogni tipo di errore senza doverli controllare all'interno dell'handler.

Altra utilità offerta dalla classe è il typing dei parametri della richiesta basati sulle api specificate.

Listing 2: Definizione API

```
export const LoginApiRoute = new Route<
  ...
>({
  method: 'post',
  path: '/login',
  schema: LoginApi.Request.Schema,
  handler: async (req, res) => {
    const token = await
       authController.login(req.body.username,
       req.body.password)
    res.sendResponse(new LoginApi.Responses.Success(token))
  },
  exceptions: [
    {
      exception: InvalidUsernameOrPassword,
      onException: (e, req, res) => {
        res.sendResponse(new UsernameOrPasswordIncorrect())
      },
    },
  ],
})
```

4.9. Definizione dei Controller

Per interfacciarsi con gli endpoint del backend sono quindi stati realizzati i **Controller** lato frontend, che incapsulano la gestione delle API fruttando il modulo opportuno per ottenere un typing delle richieste e delle risposte.

Listing 3: Definizione Controller

```
export class AuthControllerImpl extends AxiosController
    implements AuthController {
  async register(request: RegisterApi.Request.Type):
      Promise<RegisterApi.Response> {
    const body = request as RegisterApi.Request.Body
    return await
        this.post<RegisterApi.Response>('/auth/register',
        body)
  }

  async login(request: LoginApi.Request.Type):
      Promise<LoginApi.Response> {
    const body = request as LoginApi.Request.Body
    return await this.post<LoginApi.Response>('/auth/login',
        body)
  }

  async logout(): Promise<LogoutApi.Response> {
    return await
        this.post<LogoutApi.Response>('/auth/logout', {})
  }

  async refreshToken(): Promise<RefreshTokenApi.Response> {
    return await this.post<RefreshTokenApi.Response>(
        '/auth/refresh-token', {})
  }
}
```

4.10. Monitoring

Per monitorare lo stato dei servizi, è stato esposto un endpoint che restituisce un oggetto JSON contenente lo stato corrente e l'orario dell'ultimo aggiornamento. Per garantire l'aggiornamento continuo del client rispetto a queste informazioni, si è optato per l'implementazione di un meccanismo di polling dal client al server. Questa scelta consente al client di mantenere costantemente aggiornato lo stato dei servizi, anche nel caso in cui il server o i singoli microservizi diventino temporaneamente indisponibili. In questo modo, il comportamento del client rimarrà invariato, assicurando una sincronizzazione continua con lo stato più recente dei servizi monitorati. Inoltre, il microservizio di monitoraggio effettua anch'esso polling nei confronti degli altri microservizi per verificare se riceve risposta o meno.

Listing 4: Monitoring Polling - Client

```
onMounted(async () => {
  await monitoringStore.refreshServicesStatus()
  setInterval(monitoringStore.refreshServicesStatus, 2000)
})
```

4.11. Recap tecnologie utilizzate

Di seguito un elenco di tutte le tecnologie utilizzate all'interno del progetto:

4.11.1. Infrastruttura

- *Docker* per il deploy dei container.
- *Traefik* come Api-Gateway.
- *RabbitMQ* come Broker.

4.11.2. Microservizio

- *NodeJS* come runtime environment.
- *Typescript* come linguaggio di sviluppo.

- *Express* per lo sviluppo dei Webserver.
- *MongoDB* come database.
- *Mongoose* come libreria per l'utilizzo di database Mongo.
- *JWT* per la gestione dell'autenticazione.
- *Jest* come framework di Unit testing.

4.11.3. Frontend

- *Vue.js* per lo sviluppo della Single Page Application.
- *Pinia* per la gestione degli store.
- *Quasar* per la realizzazione dei componenti grafici.

4.11.4. Comunicazione

- *Axios* per la gestione di richieste HTTP.
- *Socket.io* per le comunicazioni real-time.
- *WebRTC* per le videochiamate.

4.11.5. Documentazione

- *Swagger* per la documentazione delle API.
- *AsyncAPI* per la documentazione dei messaggi infra-servizio.

5. Autovalutazione / Validazione

5.1. Struttura del repository e analisi statica

Il progetto è stato sviluppato adottando una struttura `mono repository - multi project`. Per garantire la coerenza, la qualità e la manutenibilità del codice sorgente, durante lo sviluppo abbiamo seguito **GitFlow** e adottato fin da subito diversi strumenti per l'analisi statica. Inoltre, al fine rendere estendibile e meno prolissa la manutenibilità di ogni progetto, sono stati adottati meccanismi di estensione nei file di configurazione.

5.1.1. Lint

Come strumento di Linting abbiamo adottato *ESLint*[9], grazie al quale, il nostro team ha potuto identificare e correggere errori, migliorare la coerenza del codice e rispettare le best practices di programmazione durante il processo di sviluppo.

5.1.2. Prettier

Prettier[10] è uno strumento per la formattazione del codice, integrato nel nostro flusso di lavoro per garantire uno stile uniforme in tutto il progetto. Le principali considerazioni includono:

- **Configurazione condivisa:** lo strumento è configurato per seguire uno stile condiviso in tutto il progetto, garantendo coerenza nella formattazione del codice.
- **Integrazione con ESLint:** la configurazione di Prettier è allineata con quella di ESLint, evitando conflitti e assicurando una formattazione coerente durante l'analisi statica.

L'utilizzo di Prettier ha migliorato la leggibilità del codice e semplificato notevolmente la gestione dello stile del codice degli artefatti.

[9] `https://eslint.org`
[10] `https://prettier.io`

5.1.3. Pre-commit Hook (Git)

Gli *hooks* di Git[11] sono strumenti che permettono di eseguire automaticamente azioni specifiche al verificarsi di eventi.

È stato utilizzato l'hook di *pre-commit*, al fine di eseguire controlli di formattazione prima di eseguire un commit. Questo ha permesso di migliorare la qualità del codice, evitando commit con formattazione non conforme.

5.2. Testing

La fase di testing è strutturata per verificare l'integrazione tra l'intero microservizio e le sue dipendenze, tra cui il Database e il Broker.

5.2.1. Jest

Ogni microservizio definisce degli *Unit Testing* mediante *Jest* in modo da verificare la correttezza delle richieste e delle relative risposte, dei singoli microservizi.

In questo modo, siamo stati in grado di concentrarci sui seguenti aspetti durante i test:

- Verifica della correttezza delle richieste fornite.
- Verifica della correttezza delle risposte fornite da diverse richieste.
- Verifica della correttezza che gli endpoint si comportino come atteso, controllando i *side effect* di una richiesta, eseguendo richieste successive.

Questo approccio ci ha permesso di sviluppare test solidi e garantire che i microservizi rispettassero gli standard richiesti.

5.2.2. Test sui Microservizi

Tutti i microservizi posseggono una suite di test. Di seguito un esempio con il core del testing del servizio degli utenti:

[11] `https://git-scm.com/book/it/v2/Customizing-Git-Git-Hooks`

Listing 5: microservice Test

```
const userMicroservice: Microservice = new
    Microservice(UserServiceConfiguration)
let request: supertest.SuperTest<supertest.Test>

beforeAll(async () => {
  await userMicroservice.start()
  request = supertest(userMicroservice.getServer())
})

afterAll(async () => {
  await userMicroservice.stop()
})

afterEach(async () => {
  await userMicroservice.clearDatabase()
})

describe('Register', () => {
  it('A user must provide username, password and email',
      async () => {
    let response = await request
      .post('/auth/register')
      .send({ username: 'test', password: 'test' })
    expect(response.status).toBe(400)
    // other test stuff
    response = await request
      .post('/auth/register')
      .send({ username: 'test', password: 'test', email:
          'test' })
    expect(response.status).toBe(200)
  })
    // other test stuff
})
```

5.2.3. Esecuzione dei test

Al fine di eseguire le suite di test è necessario rendere disponibili le dipendenze dei microservizi. Procedere come segue:

```
# Un terminale (directory = projectRoot)
npm i
cd dev
./runDev.sh

# Altro terminale (e.g. test su microservizio users)
npm run --workspace services/users test
# oppure (per eseguire tutti i test)
npm run test
```

5.3. Continuous Integration

Il progetto è ospitato in due servizi di hosting: GitHub e GitLab. Il primo servizio è stato utilizzato per l'intera fase di sviluppo, mentre il secondo meramente per eseguire una copia all'interno del repository fornito per il progetto.

5.3.1. Github Action

Le *GitHub Actions* sono un sistema di automazione integrato direttamente nella piattaforma *GitHub*, il servizio di hosting scelto per ospitare la fase di sviluppo del progetto.

Sono stati realizzati i seguenti workflow:

- **Check code style and lint:** il workflow esegue i controlli di analisi statica sul codice per ogni push nel repository.
- **Deploy pages:** il workflow esegue il deploy del sito statico mediante *GitHub Pages*. All'interno del sito è possibile trovare le varie api, relazione, etc.
- **Dispatch tests:** il workflow esegue i test di tutti i microservizi.

- **Testing services:** il workflow esegue i test dei microservizi che sono stati modificati nel push che lo ha innescato.

I test automatici eseguiti all'interno dei workflows necessitano delle dipendenze sopra citate, tra cui Database e Broker. Al fine di risolverle, vengono utilizzati i `services` delle GitHub Action, che permettono di istanziare gli opportuni servizi.

5.3.2. Dispatch tests e Testing services

Nell'ottica di uno sviluppo coadiuvato da un elevato numero di commit, l'idea iniziale per cui è stato realizzato il workflow *Testing services* era per evitare di eseguire l'intera suite di test ad ogni push. Infatti, il workflow esegue i test del microservizio modificato. Il supporto fornito dalle API delle GitHub Action è limitato, infatti si è dovuto scrivere un complesso script bash.

A seguito dello sviluppo è stato constatato che ciò non era necessario questo workflow aggiuntivo, infatti, l'intera suite di test non richiede eccessivo tempo per essere eseguita.

6. Deployment

Per eseguire il deploy dell'applicazione abbiamo scelto di utilizzare Docker, nello specifico, l'architettura viene servita attraverso i seguenti container:

- **Frontend Service**: Container che serve il Frontend sotto forma di Single Page Application.
- **Messages Service e relativo Db**: Microservizio responsabile della gestione dei messaggi testuali dell'applicazione (invio, ricezione, ecc...).
- **Monitoring Service e relativo Db**: Microservizio responsabile del monitoraggio dello stato di tutti i microservizi.
- **Notifications Service e relativo Db**: Mircoservizio responsabile delle notifiche degli utenti. Inoltre, detiene l'online status degli utenti del sistema.
- **Piperchat Service e relativo Db**: Microservizio responsabile della gestione dei Server e dei Canali dell'applicazione (creazione, modifica, partecipanti, ecc...).
- **Users Service e relativo Db**: Microservizio responsabile della gestione degli utenti dell'applicazione (login, registrazione, amicizie, ecc...).
- **Multimedia Service e relativo Db**: Microservizio responsabile della gestione delle chiamate infra-utenti e dei canali multimediali.
- **Broker**: Container che ospita un server di *RabbitMQ* per permettere lo scambio di messaggi all'interno del sistema
- **Coturn**: Container che ospita il server *TURN* (`https://github.com/coturn/coturn`)
- **Gateway**: Container che ospita l'*API Gateway* realizzato con *Traefik*.
- **Inspector**: Container di *utility* per debuggare e/o ispezionare i servizi dall'interno della rete.

Per ogni microservizio quindi, viene effettuato il deploy di due diversi container, uno per il Webserver, mentre l'altro per il relativo database non relazionale.

Ogni microservizio possiede le seguenti reti docker:

- **Frontend**: Rete necessaria per ricevere le richieste dal frontend tramite il Gateway.
- **Backend**: Rete necessaria per comunicare internamente con gli altri microservizi (nel nostro caso tramite message broker).
- **Microservicename-network**: Rete interna del microservizio, utile alla comunicazione con il proprio database.

6.1. Build dell'immagine Docker di PiperChat

Si è deciso di realizzare un'immagine Docker unica per l'intero progetto. Per eseguire successivamente i vari servizi è possibile utilizzare tale immagine e modificare opportunamente il comando iniziale.

Di seguito è riportato un esempio per utilizzare l'immagine di `piperchat`:

```
docker build . --tag piperchat
docker run piperchat npm run --workspace ./services/users start
```

6.2. Deploy singolo Microservizio

Per il deploy di ogni singolo microservizio è stato scritto un file di `Docker Compose`. Esso predispone l'ambiente necessario al container (reti, labels, volumi, etc.) per essere eseguito correttamente. Successivamente viene utilizzata l'immagine precedentemente creata, con il comando sovrascritto opportunamente per lo specifico microservizio.

Esempio di Docker compose relativo al singolo microservizio:

Listing 6: microservice Test

```
services:
  piperchat-service:
    image: piperchat
    command: [
        'npm',
        'run',
        '--workspace',
```

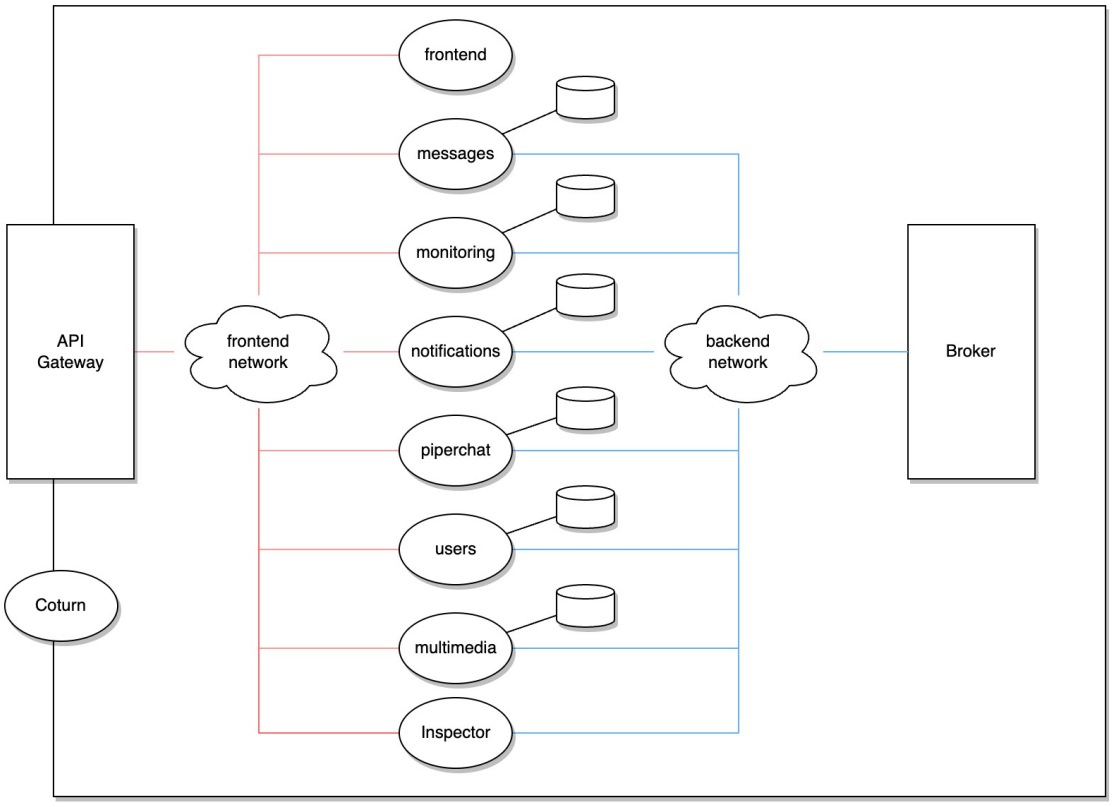
```
        './services/piperchat',
        'start'
    ]
    expose:
      - '${PIPERCHAT_SERVICE_PORT}'
    depends_on:
      db-piperchat-service:
        condition: service_healthy
      broker:
        condition: service_healthy
    networks:
      piperchat-network:
      backend:
        aliases:
          - ${PIPERCHAT_SERVICE_NAME}
      frontend:
    environment:
      - PORT=${PIPERCHAT_SERVICE_PORT}
      - AMQP_URI=${BROKER_URI}
      - MONGO_URI=
        mongodb://db-piperchat-service:27017/piperchat
    labels:
      - |
        traefik.http.routers.piperchat-service.rule=
        (Method(`GET`, `POST`) && Path(`/servers`)) ||
            ... other paths

db-piperchat-service:
    image: mongo
    expose:
      - '27017'
    volumes:
      - './.docker/db-piperchat:/data/db'
    healthcheck:
      test: |
        host=`hostname --ip-address || echo '127.0.0.1'`;
        mongo --quiet $${host}/test --eval
        'quit(db.runCommand({ ping: 1 }).ok ? 0 : 2)'
```

```
46          && echo 0 || echo 1
47      networks:
48        - piperchat-network
49
50  networks:
51    piperchat-network:
```

6.3. Deploy dell'architettura

Come accennato in precedenza, per ogni microservizio è stato scritto un apposito file di `Docker Compose`. L'unione di tutti i `docker-compose.yaml` è stata automatizzata tramite uno script bash, che permette di operare sull'intera architettura, aggregando precedentemente i file di tutti i microservizi. Lo script é `./composeAll.sh`.

Di seguito è riportata l'intera architettura di cui viene eseguito il deploy, con tutti i microservizi, i relativi database, le reti, il broker ed il gateway.



6.4. Testing deploy

Per effettuare il testing lato Backend, è stato realizzato un ulteriore `Docker Compose` il quale ci ha permesso deployare un infrastruttura "alleggerita", dotata unicamente dei container necessari a testare i singoli microservizi, nonché:

- Un'istanza del Broker
- Un'istanza del database
- Un'istanza di Mongo Express per visualizzare il database

Questo `Docker Compose` è situato nella cartella `/dev`, insieme allo script che ne automatizza l'esecuzione, chiamato `runDev.sh`.

Di seguito riportato il `Docker Compose` dell'infrastruttura usata per il testing:

Listing 7: Testing Deploy

```
services:
  db-service:
    image: mongo
    ports:
      - '27017:27017'
    healthcheck:
      test: |
        host=`hostname --ip-address || echo '127.0.0.1'`;
        mongo --quiet $${host}/test --eval
            'quit(db.runCommand({ ping: 1 }).ok ? 0 : 2)'
            && echo 0 || echo 1

  broker:
    image: rabbitmq:3-management-alpine
    ports:
      - '5672:5672'
      - '15672:15672'
    healthcheck:
      test: ['CMD', 'rabbitmq-diagnostics', '-q', 'ping']

  mongo-express:
    image: mongo-express:latest
```

```
23      restart: always
24      ports:
25        - '8081:8081'
26      environment:
27        ME_CONFIG_MONGODB_SERVER: 'db-service'
28      depends_on:
29        db-service:
30          condition: service_healthy
```

6.5. Istruzioni per il Deployment

Di seguito, elencate le istruzioni per eseguire il deploy dell'applicazione:

1. Clone del Repository:

   ```
   $ git clone https://github.com/zucchero-sintattico/piperchat.git
   ```

2. Installazione delle dipendenze:

   ```
   $ npm i
   ```

3. Copia del file contenente le variabili di environment dal template:

   ```
   $ cp .env.template .env
   ```

4. Eseguire il deploy dell'architettura tramite lo script che esegue anche il build dell'immagine:

   ```
   $ ./cleanDeploy.sh
   ```

5. Dalla seconda volta in poi, per rieseguire il deploy dell'architettura mantenendo i volumi e le immagini precedentemente create, basterà lanciare lo script:

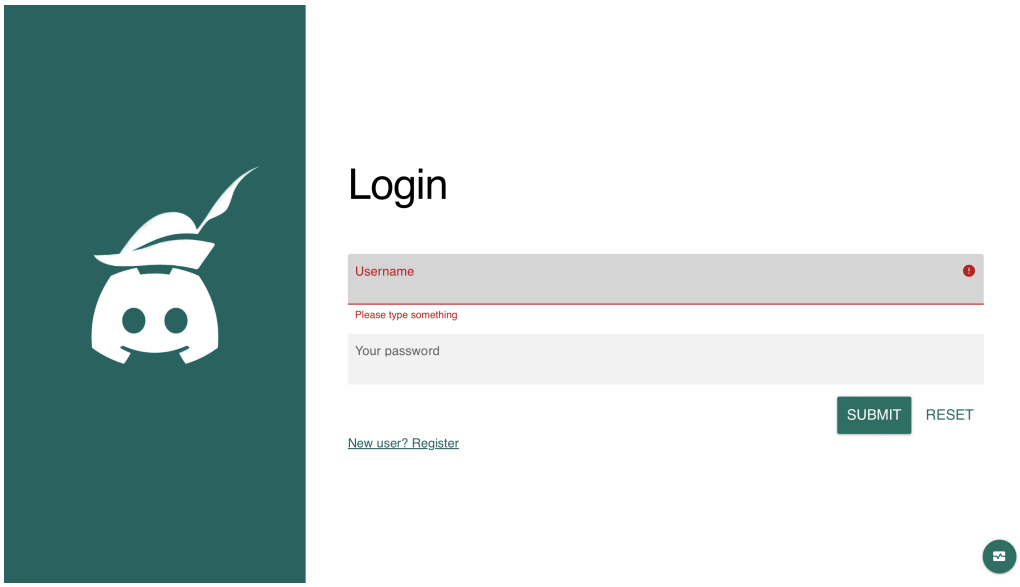
   ```
   $ ./deploy.sh
   ```

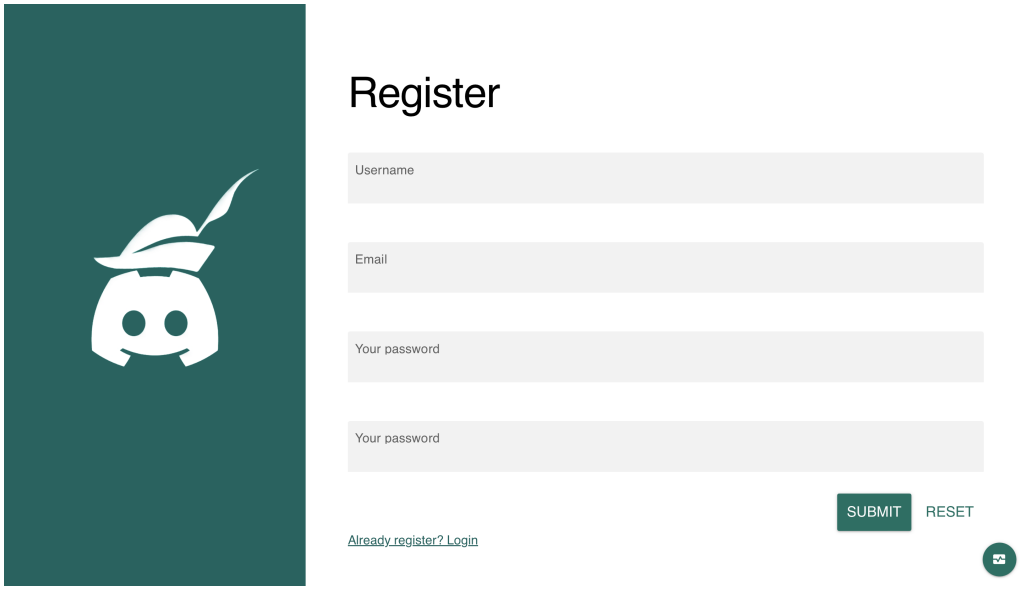
7. Esempi d'utilizzo

Di seguito vengono riportati alcuni esempi di utilizzo della nostra applicazione:

7.1. Login/Registrazione

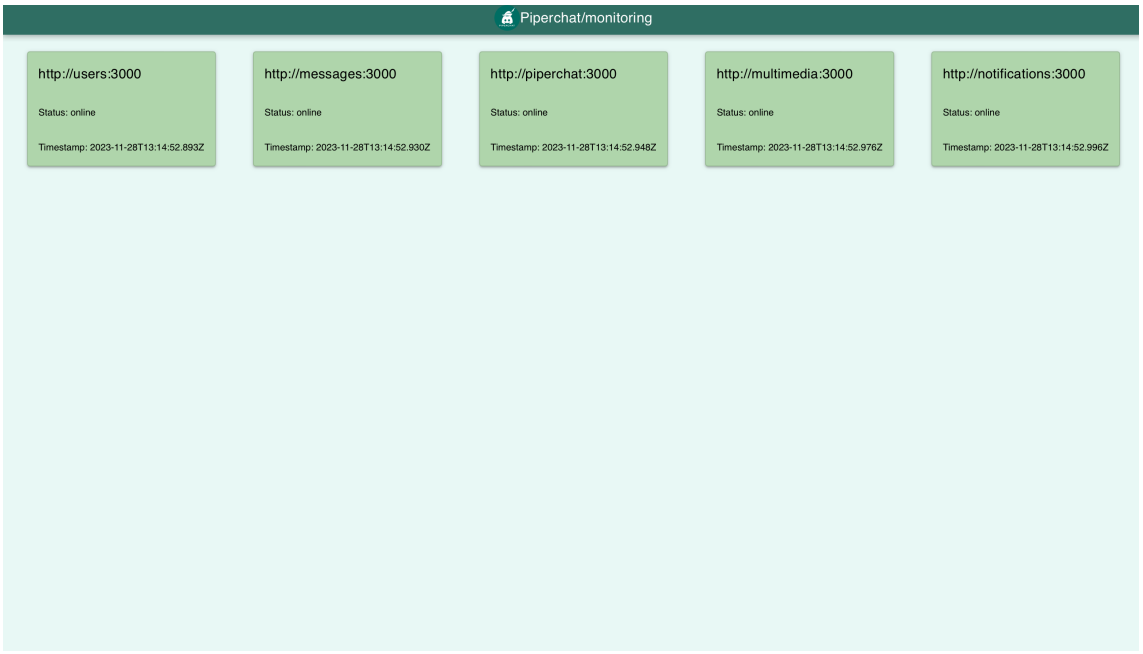
L'accesso a questa sezione non richiede alcuna autenticazione preliminare. Gli utenti hanno la possibilità di effettuare la registrazione o il login per accedere alle funzionalità offerte.





7.2. Monitoring/Healthcheck

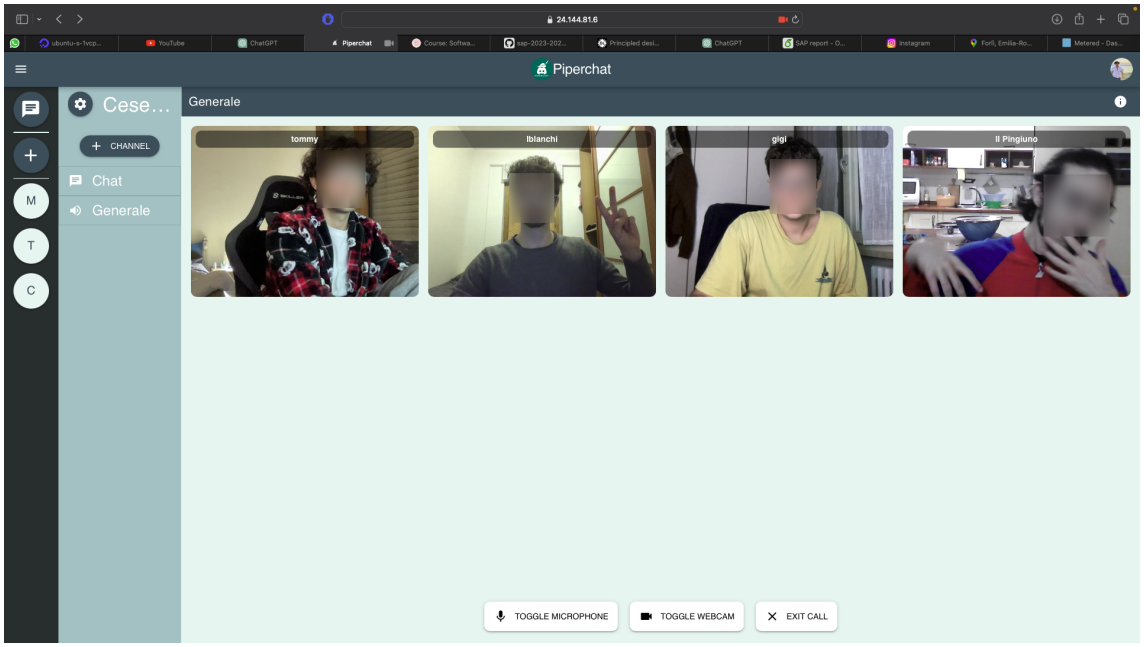
L'accesso a questa sezione è libero e non richiede credenziali di accesso. Qui gli utenti possono monitorare lo stato dei microservizi in modo rapido e semplice.



7.3. HomePage

Per accedere a questa sezione è necessario eseguire l'accesso. In caso di assenza di un access token valido, verrà automaticamente reindirizzati alla pagina di login. La struttura della pagina è articolata in due sezioni principali:

- **Menù laterale**: Da qui è possibile navigare tra server, canali e messaggi diretti.
- **Area principale**: Questa sezione mostra i contenuti delle chat e delle video-chiamate.



8. Conclusioni

Il progetto Piperchat è stato sviluppato con l'obiettivo di creare una piattaforma di comunicazione ispirata a Discord, offrendo agli utenti la possibilità di interagire in varie forme, creare connessioni sociali e gestire server personalizzati. L'architettura a microservizi è stata scelta per gestire la complessità del sistema, consentendo una maggiore coesione e scalabilità.

Durante lo sviluppo, sono stati affrontati diversi aspetti, tra cui la gestione delle amicizie, la messaggistica, la creazione e gestione di server e canali multimediali. L'implementazione di funzionalità come notifiche, chiamate vocali e video ha arricchito l'esperienza dell'utente.

La valutazione della qualità del software prodotto ha considerato aspetti cruciali come funzionalità, affidabilità, scalabilità, sicurezza, usabilità e prestazioni. L'approccio a microservizi ha dimostrato la sua efficacia nel garantire una struttura modulare e gestibile.

8.1. Sviluppi futuri

A causa delle scarse tempistiche, l'implementazione della dashboard relativa ai log dell'intera applicazione, è stata messa in secondo piano. Detto ciò dunque, la prima feature da realizzare in termini di sviluppi futuri sarebbe proprio quest'ultima.

Altro miglioramento che potremmo apportare alla nostra applicazione, riguarderebbe il deploy dell'architettura, difatti attualmente, l'applicazione di videochat viene distribuita utilizzando Docker Compose con un approccio basato su una singola macchina. Tuttavia, per migliorare la scalabilità, la resilienza e la gestione delle risorse, è previsto un futuro passaggio a un'architettura basata su Docker Swarm.

8.2. Cosa abbiamo imparato

Durante lo sviluppo di Piperchat, abbiamo acquisito una comprensione approfondita dell'architettura a microservizi e delle sfide associate. Abbiamo imparato a bilanciare aspetti cruciali come la sicurezza, la scalabilità e l'usabilità in un progetto di comunicazione in tempo reale.

La collaborazione tra diversi team per lo sviluppo dei singoli microservizi e la gestione di comunicazione tra di essi attraverso un message broker hanno ampliato la nostra conoscenza delle best practice nello sviluppo di sistemi distribuiti.

Inoltre, l'importanza di valutare la qualità del software in termini di funzionalità, affidabilità e prestazioni ci ha fornito un quadro completo delle sfide e delle opportunità nell'implementazione di una piattaforma di comunicazione complessa come Piperchat.

A. Benchmarking

Di seguito sono riportate le prove di *benchmarking* effettuate, al fine di mettere sotto sforzo il sistema per verificare che la *scalabilità orizzontale* porti parte dei risultati attesi: aumentare il numero di richieste gestite nell'arco di un determinato periodo.

Obiettivo: il test consiste nell'eseguire lo stesso carico di lavoro lato client variando il numero di repliche del servizio che le deve gestire.

Lo scenario prevede un client che effettua richieste HTTP verso il server, dove è in esecuzione il software di Piperchat, all'interno degli appositi container docker.

A.1. Configurazione

Sono state utilizzate le seguenti macchine:

Componente	**Client**	**Server**
S.O.	MacOS 14.1	Ubuntu 23.10
CPU	Apple M1 Pro	Intel Core i7-8700 6 core
RAM	16 GB	16 GB
Connessione	Connessione LAN @ ∼500 Mbit/s	

Lo strumento utilizzato per effettuare le richieste HTTP lato client è *wrk*, un software di benchmarking che permette di generare carichi di lavoro significativi, sfruttando una singola CPU multi-core.

Il microservizio sotto osservazione è *users-service*, del quale è stata testata la rotta `/auth/login` con le credenziali di un utente già registrato, attraverso la seguente configurazione:

```
$ wrk -t10 -c150 -d30s http://<server-ip>/auth/login -s ./post.lua
```

```
// Configurazione script del software (post.lua)
wrk.method = "POST"
wrk.body = '{"username": "user", "password": "12341234"}'
wrk.headers["Content-Type"] = "application/json"
```

- `-t10`: Vengono impiegati 10 threads
- `-c150`: Vengono simulate 150 connessioni attive (utenti)
- `-d30s`: Il test dura 30 secondi

A.2. Il test

Il test è stato eseguito più volte, mediante la configurazione esposta in precedenza, variando il numero di repliche del servizio.

Di seguito vengono riportati i risultati, ognuno dei quali è la media di 4 esecuzioni.

Tabella 2: 1 Replica

Thread Stats	**Avg**	**Stdev**	**Max**	**+/- Stdev**
Latency	122.43ms	87.88ms	1.99s	98.19%
Req/Sec	125.04	27.36	290.00	84.05%
Requests	37424			
Data Read	9.17MB			
Socket Timeout	49			

Tabella 3: 2 Repliche

Thread Stats	**Avg**	**Stdev**	**Max**	**+/- Stdev**
Latency	88.17ms	113.72ms	1.84s	97.79%
Req/Sec	202.99	35.28	350.00	70.80%
Requests	60708			
Data Read	14.88MB			
Socket Timeout	20			

Tabella 4: 3 Repliche

Thread Stats	Avg	Stdev	Max	+/- Stdev
Latency	71.24ms	108.48ms	1.87s	96.68%
Req/Sec	255.34	45.02	460.00	71.90%
Requests	76351			
Data Read	18.71MB			
Socket Timeout	10			

Tabella 5: 5 Repliche

Thread Stats	Avg	Stdev	Max	+/- Stdev
Latency	61.14ms	113.39ms	1.87s	96.97%
Req/Sec	332.08	56.51	560.00	74.39%
Requests	100839			
Data Read	24.28MB			
Socket Timeout	6			

Tabella 6: 10 Repliche

Thread Stats	Avg	Stdev	Max	+/- Stdev
Latency	43.87ms	29.17ms	413.79ms	88.57%
Req/Sec	370.84	74.67	545.00	80.13%
Requests	110072			
Data Read	26.98MB			
Socket Timeout	0			

Tabella 7: 20 Repliche

Thread Stats	Avg	Stdev	Max	+/- Stdev
Latency	137.94ms	213.60ms	1.98s	91.61%
Req/Sec	162.18	108.81	420.00	58.31%
Requests	15308			
Data Read	3.52MB			
Socket Timeout	550			

A.3. Risultati

Il test effettuato ha portato ai risultati attesi. I dati di maggior rilievo da prendere che possono essere osservati sono la *latenza* ed il *numero di richieste*.

Come si può notare, il numero di richieste tendere ad aumentare finché non si arriva al numero di repliche uguale al numero di core effettivi della macchina che deve prendere in carico la richiesta, per poi stabilizzarsi.

Invece, la latenza di risposta alle richieste tende a diminuire costantemente finché non si raggiunge il numero di threads[12] di cui dispone la macchina.

Superati i numeri evidenziati in precedenza, le prestazioni iniziano a degradare. Come si può notare in Tabella 7, il numero di socket che non riesce ad effettuare una connessione tende ad aumentare notevolmente.

[12]Intel® Hyper-Threading Technology

B. Tentativo di deployment reale

Abbiamo proceduto con il deploy dell'applicazione online con l'obiettivo di testare e utilizzare concretamente l'applicativo. Durante questo processo, sono emersi alcuni problemi rilevanti relativi all'implementazione delle WebSocket.

B.1. Problema con WebSocket non Sicure

Durante il deploy, ci siamo accorti che l'applicazione utilizzava WebSocket non sicure (ws) invece di WebSocket sicure (wss). Questo ha causato il blocco delle connessioni da parte dei browser, poiché molti moderni browser richiedono connessioni sicure per proteggere la privacy degli utenti.

Per affrontare questo problema, abbiamo implementato una soluzione intermedia utilizzando Nginx come proxy SSL. Nginx è stato configurato per gestire la crittografia SSL/TLS delle connessioni WebSocket, consentendo l'utilizzo di connessioni sicure (wss). Questo approccio ci ha permesso di ottenere connessioni sicure senza dover apportare modifiche dirette al codice sorgente dell'applicazione.

```
location / {
    # Configurazione del proxy pass verso il server sulla porta 80
    proxy_pass http://localhost:80;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}

location /notification {
    proxy_pass http://localhost:80;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}

location /webrtc {
```

```
    proxy_pass http://localhost:80;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

C. Sperimentale: deploy con Dev Containers

A fini sperimentali, è stato eseguito il deploy dell'infrastruttura attraverso la tecnologia *Dev Containers* di GitHub.

Di seguito sono riportate le istruzioni per eseguire il deploy:

1. Fork del repository
2. Dalla pagina di GitHub

   `Code > Codespaces > Create codespace on main`

3. Attendere la creazione dell'ambiente.
4. Utilizzare il servizio tramite il link fornito dall'ambiente di codespace.
5. (Opzionale) Modificare la visibilità delle porte tramite l'interfaccia grafica nell'apposita sezione, in modo da permettere di utilizzare il sistema anche a terzi, tramite il link fornito.