

# Documentazione Whisperamm

[Web-based Real-Time Multiplayer Game: Whisperamm](#)

[Whisperamm: cos'è?](#)

[Analisi dei Requisiti](#)

[Tabella 1: Requisiti di Accesso e Gestione Lobby](#)

[Tabella 2: Requisiti logica di Game](#)

[Use Case Diagram](#)

[Attori](#)

[Activity Diagram](#)

[Activity Diagram: Creazione e Avvio Partita \(Fase Lobby\)](#)

[Activity Diagram: Flusso di Gioco \(Round, Dadi e Vittoria\)](#)

[Stack Tecnologico](#)

[Tecnologie Software](#)

[Architettura e Design del Sistema](#)

[Vista ad Alto Livello del Sistema - Context Diagram](#)

[Pattern Architettonico per il Back-end](#)

[Package Diagram](#)

[Modellazione dei Dati](#)

[State Machines](#)

[Ciclo di Vita della Room](#)

[Flusso di Gioco \(Game Loop\)](#)

[Struttura Front-End](#)

[Dettaglio delle Interazioni](#)

[Implementazione](#)

[Entry point Front-end: App.jsx e providers](#)

[Registrazione](#)

[Home](#)

[Lobby](#)

[Game](#)

[phaseWord](#)

[phaseVoting](#)

[Janus Media Server](#)

[Deploy](#)

[Soluzione Architetturale: SSL Termination Proxy](#)

## Web-based Real-Time Multiplayer Game: Whisperamm

A project made by:

Luigi Caretti M63001721

Giuseppe Castaldo M63001741

Pasquale Paolo Silvenni M63001717

## Whisperamm: cos'è?

Il presente documento illustra il lavoro svolto per la realizzazione di **Whisperamm**, una piattaforma web interattiva per il gioco multiplayer in tempo reale di *social deduction* ispirato alle meccaniche di "Mr. White" e "Undercover", dunque la **competizione** tra due fazioni:

- **Civili:** utenti che ricevono la stessa parola segreta.
- **Impostori:** utenti che ricevono una parola *leggermente diversa* ma contestualmente affine a quella dei civili.

La sfida centrale del gioco consiste nell'individuare ed eliminare gli impostori basandosi sulle descrizioni vocali fornite dai partecipanti, le quali devono essere abbastanza vaghe da non rivelare la parola agli avversari, ma abbastanza precise da confermare la propria identità agli alleati.

## Analisi dei Requisiti

L'analisi dei requisiti ha l'obiettivo di definire in modo formale e non ambiguo le funzionalità che il sistema **Whisperamm** deve offrire.

Per una maggiore chiarezza espositiva e progettuale, i requisiti funzionali sono stati categorizzati in tre macro-aree: **Accesso - Gestione Lobby** e **Logica di Gioco**.

**Tabella 1: Requisiti di Accesso e Gestione Lobby**

Questa sezione copre le funzionalità preliminari necessarie per l'identificazione dell'utente e la configurazione dell'ambiente di gioco.

ID	Requisito	Descrizione
R_ACC_01	<b>Registrazione Utente</b>	Il sistema deve permettere all'utente di inserire un <i>nickname</i> e associarlo a una sessione temporanea.
R_ACC_02	<b>Creazione Partita</b>	L'utente (Admin) deve poter configurare la stanza (Nome, Max Giocatori, Round).
R_ACC_03	<b>Generazione Codice</b>	Il sistema deve generare un codice univoco per l'invito alla partita.
R_ACC_04	<b>Accesso alla Partita</b>	Il sistema deve consentire l'accesso tramite codice, verificando la disponibilità di posti.
R_ACC_05	<b>Lobby di Attesa</b>	Il sistema deve mostrare in tempo reale la lista dei partecipanti connessi.
R_ACC_06	<b>Chat Lobby</b>	Il sistema deve fornire una chat testuale pre-partita per i partecipanti della lobby.
R_ACC_07	<b>Notifiche Sistema</b>	Il sistema deve notificare in chat gli ingressi e le uscite dei giocatori.
R_ACC_08	<b>Stato "Pronto"</b>	Il sistema deve permettere ai partecipanti di impostare il proprio stato su "Pronto" e visualizzare lo stato degli altri. Il pulsante "Avvia Partita" dell'Admin deve abilitarsi <b>solo quando tutti</b> i giocatori presenti sono "Pronti".

**Tabella 2: Requisiti logica di Game**

Questa sezione descrive tutte le meccaniche operative che regolano lo svolgimento della partita, dalla distribuzione dei ruoli alla determinazione del vincitore.

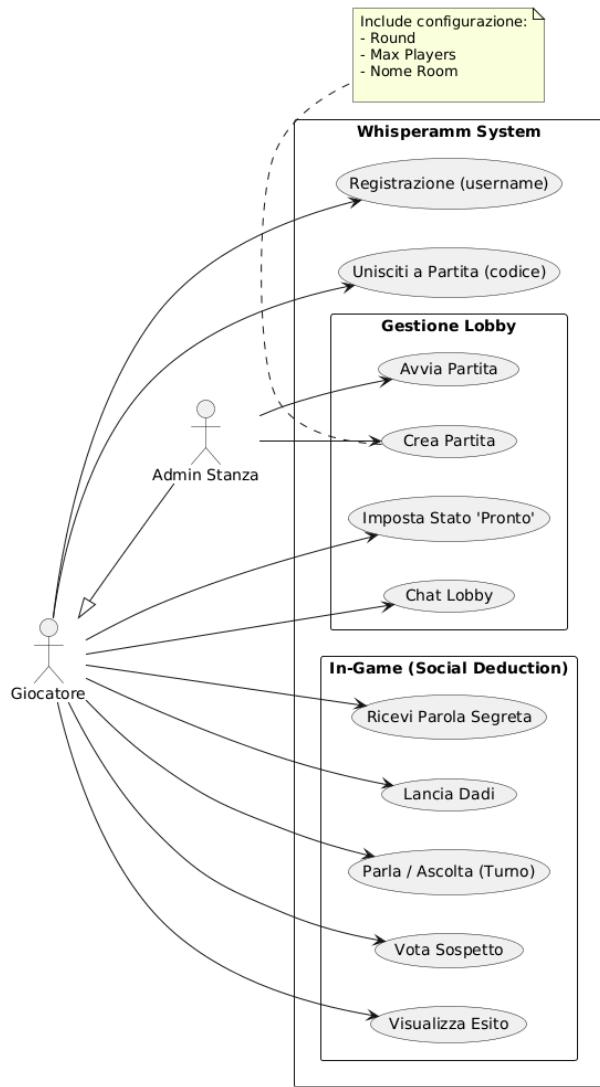
ID	Requisito	Descrizione
R_GAME_01	<b>Inizializzazione Partita</b>	Il sistema deve transire dallo stato di Lobby allo stato di Gioco su comando dell'Admin.
R_GAME_02	<b>Assegnazione Ruoli</b>	Il sistema deve assegnare casualmente i ruoli ("Civile" o "Impostore") in base alla configurazione della stanza.
R_GAME_03	<b>Distribuzione Parole</b>	Il sistema deve inviare ai client le parole segrete differenziate per ruolo (Parola A per Civili, Parola B per Impostori), garantendo la segretezza delle informazioni.
R_GAME_04	<b>Infrastruttura Audio</b>	Il sistema deve inizializzare una <i>audio room</i> tramite Media Server, permettendo la connessione tecnica dei flussi WebRTC (RTP) di tutti i partecipanti.
R_GAME_05	<b>Ordine di Gioco</b>	Il sistema deve determinare l'ordine iniziale tramite un unico lancio di dadi. Nei round successivi, l'ordine di parola deve <b>scalare (ruotare)</b> automaticamente, rendendo primo il giocatore successivo nella lista.
R_GAME_06	<b>Gestione Esclusiva Audio</b>	Durante la fase di descrizione, il sistema deve abilitare il microfono <b>esclusivamente</b> al giocatore di turno, forzando la modalità "muto" (solo ascolto) per tutti gli altri partecipanti.
R_GAME_07	<b>Timer di Turno</b>	Il sistema deve gestire un timer per il turno di parola corrente; allo scadere del tempo, deve revocare la parola al giocatore attivo e passarla al successivo in lista.
R_GAME_08	<b>Gestione Votazione</b>	Il sistema deve permettere a ogni giocatore attivo di selezionare un sospetto da eliminare durante la fase di votazione dedicata.
R_GAME_09	<b>Eliminazione e Mute</b>	Il sistema deve calcolare la maggioranza dei voti, comunicare l'eliminazione del giocatore e revocargli permanentemente i permessi di trasmissione audio per il resto della partita.
R_GAME_11	<b>Condizioni di Vittoria</b>	Il sistema deve verificare lo stato al termine di ogni fase. La partita termina se: 1. <b>Vittoria Civili:</b> l'impostore è stato eliminato. 2. <b>Vittoria Impostori:</b> è stato raggiunto il <b>Limite Round</b> senza aver eliminato l'impostore.

# Use Case Diagram

Il diagramma dei casi d'uso illustra le funzionalità offerte dal sistema agli attori principali. In **Whisperamm**, gli attori sono differenziati dai privilegi di gestione della stanza, pur partecipando entrambi al gioco.

## Attori

- **Giocatore:** L'utente generico che accede al sistema. Può registrarsi, unirsi a partite esistenti e partecipare al gameplay (parlare, votare, lanciare dadi).
- **Admin Stanza:** Una specializzazione del Giocatore. È colui che ha creato la lobby e possiede privilegi esclusivi per la configurazione e l'avvio della partita.



# Activity Diagram

Gli Activity Diagram sono stati utilizzati per modellare i flussi procedurali e la logica comportamentale del sistema.

L'analisi si concentra su due macro-processi fondamentali: la fase di configurazione e avvio (**Lobby**) e il ciclo di vita della partita vera e propria (**Game Loop**).

## Activity Diagram: Creazione e Avvio Partita (Fase Lobby)

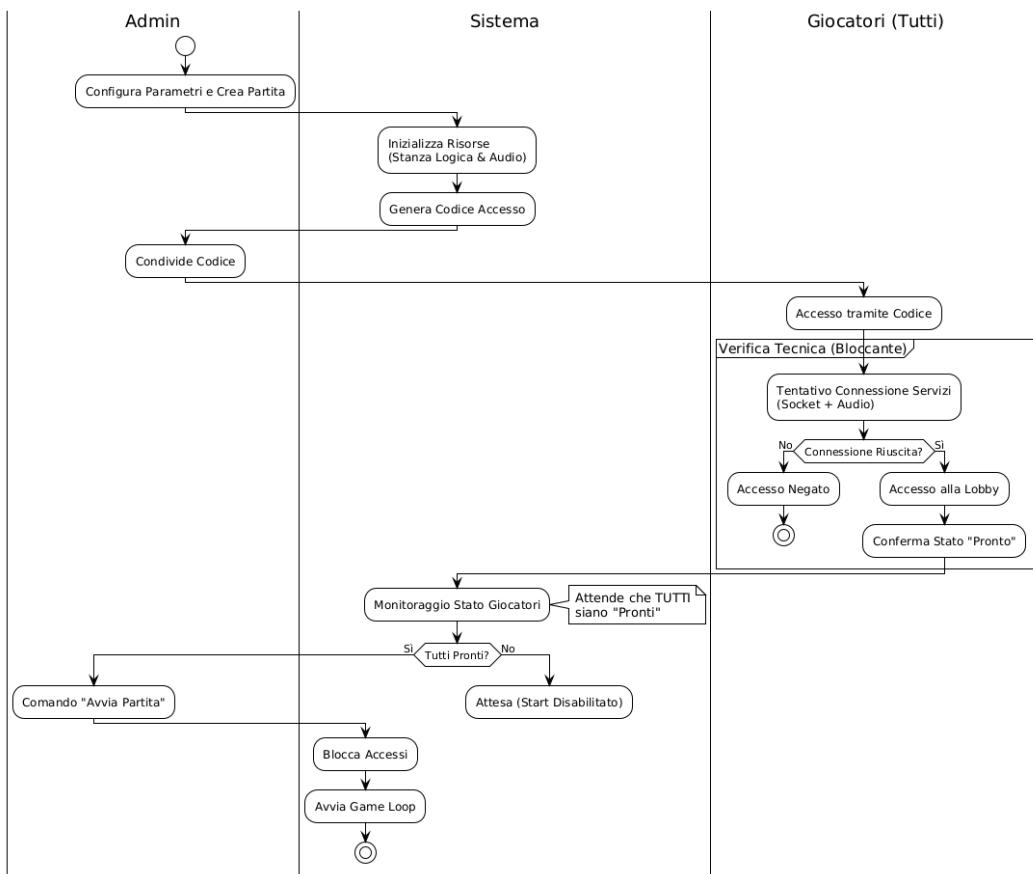
Il processo coinvolge tre attori: l'**Admin** (che orchestra), il **Sistema** (che gestisce lo stato e l'infrastruttura) e i **Giocatori** (ospiti).

Le macro-fasi individuate sono:

- **Setup e Inizializzazione:** L'Admin definisce i parametri della sessione. Il Sistema risponde allocando le risorse necessarie (stanza di gioco e room audio) e generando il codice di accesso.
- **Accesso e Verifica Tecnica:** I giocatori accedono alla stanza.  
In questa fase avviene un **check bloccante**: il sistema valida la connettività (Socket e Audio)

prima di permettere l'interazione. Solo se il sottosistema audio è operativo, l'utente viene ammesso alla Lobby.

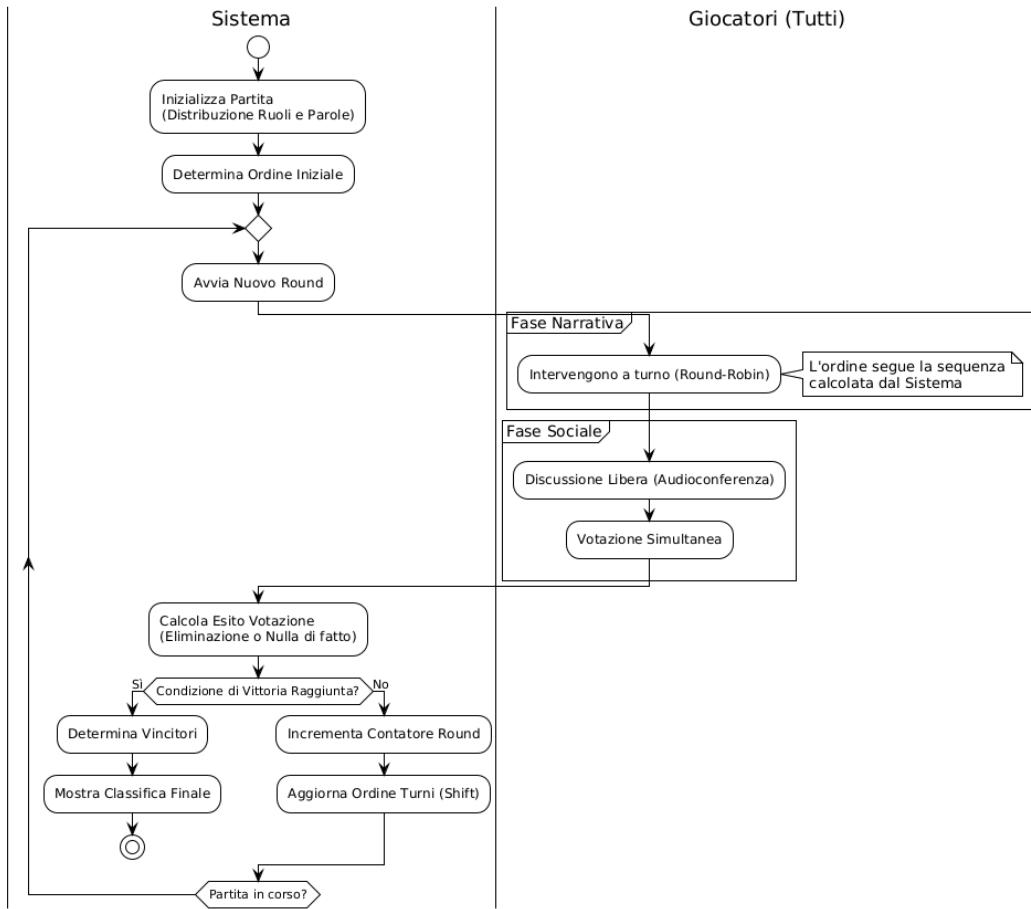
- **Sincronizzazione (Ready-Check):** Una volta ammessi, i giocatori confermano la propria presenza ("Pronto"). Il Sistema funge da barriera logica (*Gatekeeper*), abilitando il comando di avvio per l'Admin solo quando la totalità dei partecipanti ha completato il setup ed è pronta a giocare.



## Activity Diagram: Flusso di Gioco (Round, Dadi e Vittoria)

Questo diagramma descrive l'intero ciclo di vita della partita attiva.

Viene evidenziata la struttura iterativa dei **Round** che continua finché non si verifica una condizione di vittoria (Impostore scoperto o Limite Round raggiunto).



## Stack Tecnologico

Durante il ciclo di vita del progetto, sono stati impiegati diversi strumenti per supportare le fasi di sviluppo, gestione del codice e progettazione.

- **Visual Studio Code:** ambiente di sviluppo (IDE) principale.
- **Git & GitHub:** utilizzati per il versioning del codice e la collaborazione.
- **PlantUML:** utilizzato per la creazione di diagrammi UML.
- **Redis Insight:** ambiente per visualizzare i dati nel db Redis in Cloud.

## Tecnologie Software

L'architettura del sistema si divide in tre macro-aree funzionali: il Back-End (che centralizza logica, sicurezza e dati), il Front-End (interfaccia utente) e il Media Server dedicato.

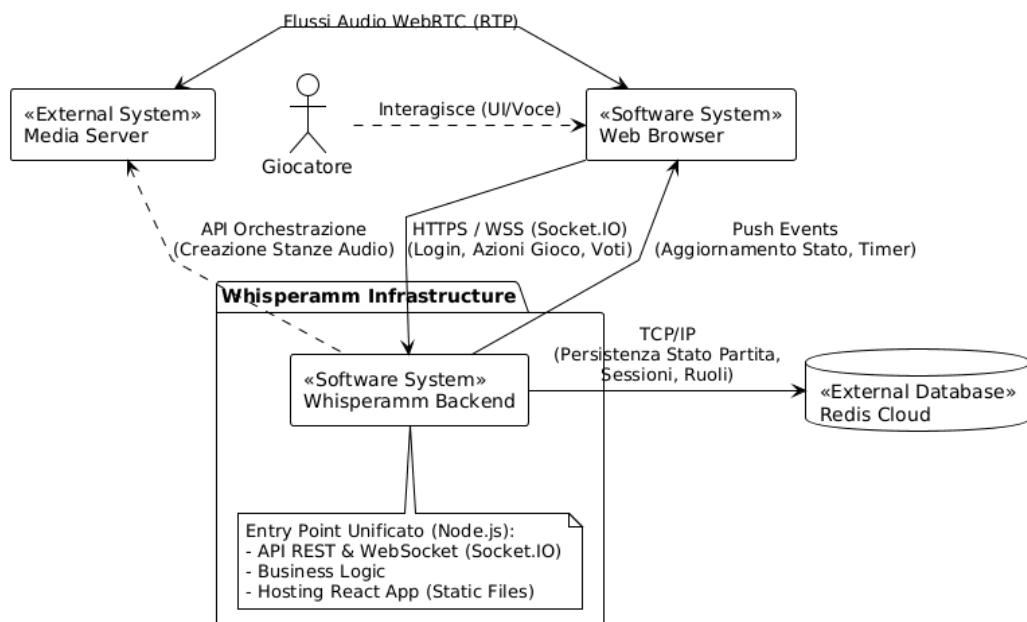
- **Back-End (Ecosistema Node.js):** Il nucleo del server è sviluppato in Node.js e orchestra diverse tecnologie per garantire performance, sicurezza e scalabilità.
  - **Web Server & API (Express):** Il framework **Express** è utilizzato per esporre le API RESTful, gestire il routing e servire direttamente i file statici della build di produzione, fungendo da entry point unico dell'applicazione.

- **Canali Bidirezionali (Socket.IO):** Integrata nello stesso server HTTP, la libreria **Socket.IO** gestisce canali di comunicazione persistenti. Viene utilizzata esclusivamente per lo scambio di messaggi JSON leggeri (segnalazione eventi di gioco, chat, cambio turni), separando la logica di gioco dallo streaming media.
- **Persistenza Dati (Redis):** Per la gestione dello stato volatile delle partite, il backend si connette a **Redis**. Essendo un database in-memory, offre tempi di accesso immediati, fondamentali per mantenere sincronizzato lo stato condiviso tra i client senza colli di bottiglia I/O.
- **Autenticazione (JWT):** La sicurezza e l'identificazione degli utenti sono gestite tramite **JSON Web Token**. Questo approccio favorisce un'architettura *stateless*, dove il server verifica la validità del token senza dover mantenere sessioni server-side pesanti.
- **Front-End (React.js):** L'interfaccia utente è una Single Page Application (SPA) realizzata con **React**, progettata per la massima reattività. Comunica con il backend tramite chiamate API per le operazioni standard e tramite eventi Socket per l'interazione di gioco in tempo reale.
- **Media Server (Janus):** La gestione dei flussi audio (audioconferenza) è disaccoppiata dal server di gioco e delegata a **Janus WebRTC Gateway**, che si occupa specificamente del routing dei pacchetti audio RTP/RTCP tra i partecipanti con latenza minima.

## Architettura e Design del Sistema

### Vista ad Alto Livello del Sistema - Context Diagram

Il diagramma illustra l'architettura logica di Whisperamm, il cui cuore pulsante è il **Backend Node.js**. Questo componente agisce come *entry point* unificato: serve l'applicazione Front-End (file statici React), espone le API REST e gestisce i canali bidirezionali **Socket.IO** per la sincronizzazione del gioco.



Il sistema interagisce con due servizi esterni:

- **Redis:** Garantisce la persistenza rapida dello stato delle partite e delle sessioni, mantenendo i dati sensibili isolati dal client.
- **Media Server (Janus):** Gestisce i flussi audio WebRTC. Nello schema architettonale, il Backend si occupa dell'orchestrazione (creazione stanze), mentre i flussi RTP viaggiano direttamente tra browser e media server.

#### **Nota sull'Implementazione e Sicurezza**

Sebbene il diagramma rappresenti l'architettura di riferimento sicura (dove è il Backend a comandare Janus), nell'attuale prototipo la logica di orchestrazione delle stanze audio è stata intenzionalmente lasciata lato Client.

Questa discrepanza implementativa espone il sistema a una specifica vulnerabilità (manipolazione non autorizzata delle VideoRoom) la cui analisi e le cui contromisure saranno discusse all'esame di Network Security.

## Pattern Architettonico per il Back-end

Il backend di Whisperamm è progettato seguendo il principio della *Separation of Concerns* (SoC), adottando una **Layered Architecture (Architettura a Livelli)**.

A differenza del pattern MVC tradizionale, dove il server è spesso responsabile anche della generazione della vista, questa architettura delega interamente la UI al client (React), focalizzandosi sulla struttura **Controller-Service-Data Access**.

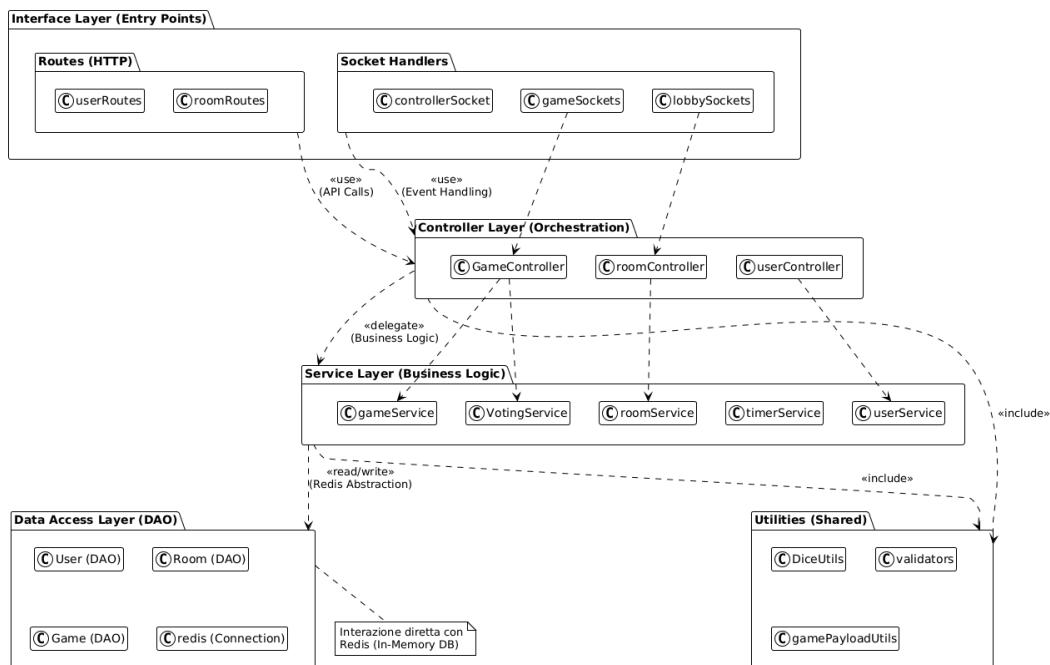
L'organizzazione del codice è suddivisa in livelli logici gerarchici:

- **Interface Layer:** rappresenta la frontiera del sistema verso il mondo esterno.  
Gestisce in parallelo due paradigmi di comunicazione:
  - **REST API ( /routes )**: definisce gli endpoint HTTP per le operazioni standard (es. Login, Creazione Stanza) e smista le richieste.
  - **Event Handlers ( /socket )**: intercetta gli eventi WebSocket real-time (es. messaggi di chat, azioni di gioco).  
Entrambi i canali convergono verso il livello successivo, normalizzando l'input.
- **Controller Layer ( /controllers )**: agisce da orchestratore.  
I controller (es. `GameController.js`, `RoomController.js`) ricevono l'input sanitizzato dal livello superiore, validano i dati e delegano l'esecuzione delle operazioni complesse al *Service Layer*. Questo livello non contiene logica di business, ma si limita a coordinare la richiesta e formattare la risposta (HTTP o evento Socket) verso il client.
- **Service Layer (Business Logic):** situato nella cartella `/services`, costituisce il cuore applicativo del sistema (es. `GameService.js`, `VotingService.js`).  
Qui risiedono: algoritmi di assegnazione ruoli, calcolo delle votazioni e gestione dei timer. I servizi sono progettati per essere **agnosticisti rispetto al protocollo**, eseguendo le operazioni senza conoscere se la richiesta provenga da una chiamata API o da un evento Socket, garantendo massima riusabilità e testabilità.
- **Data Access Layer ( /models )**: funge da strato di astrazione verso il database.  
Poiché viene utilizzato **Redis** (store key-value in-memory), i moduli in questa cartella (es. `Game.js`, `User.js`) agiscono come **Data Access Objects (DAO)**.  
Essi encapsulano le query a basso livello, offrendo ai servizi un'interfaccia pulita per leggere e scrivere lo stato volatile dei dati.

- **Utilities** (`/utils`): un livello trasversale che contiene funzioni helper (es. `DiceUtils.js`, `Validators.js`), utilizzate da controller e servizi per compiti specifici e ripetibili.

## Package Diagram

Il diagramma struttura il backend in cinque package logici principali, rispecchiando fedelmente l'organizzazione del file system:



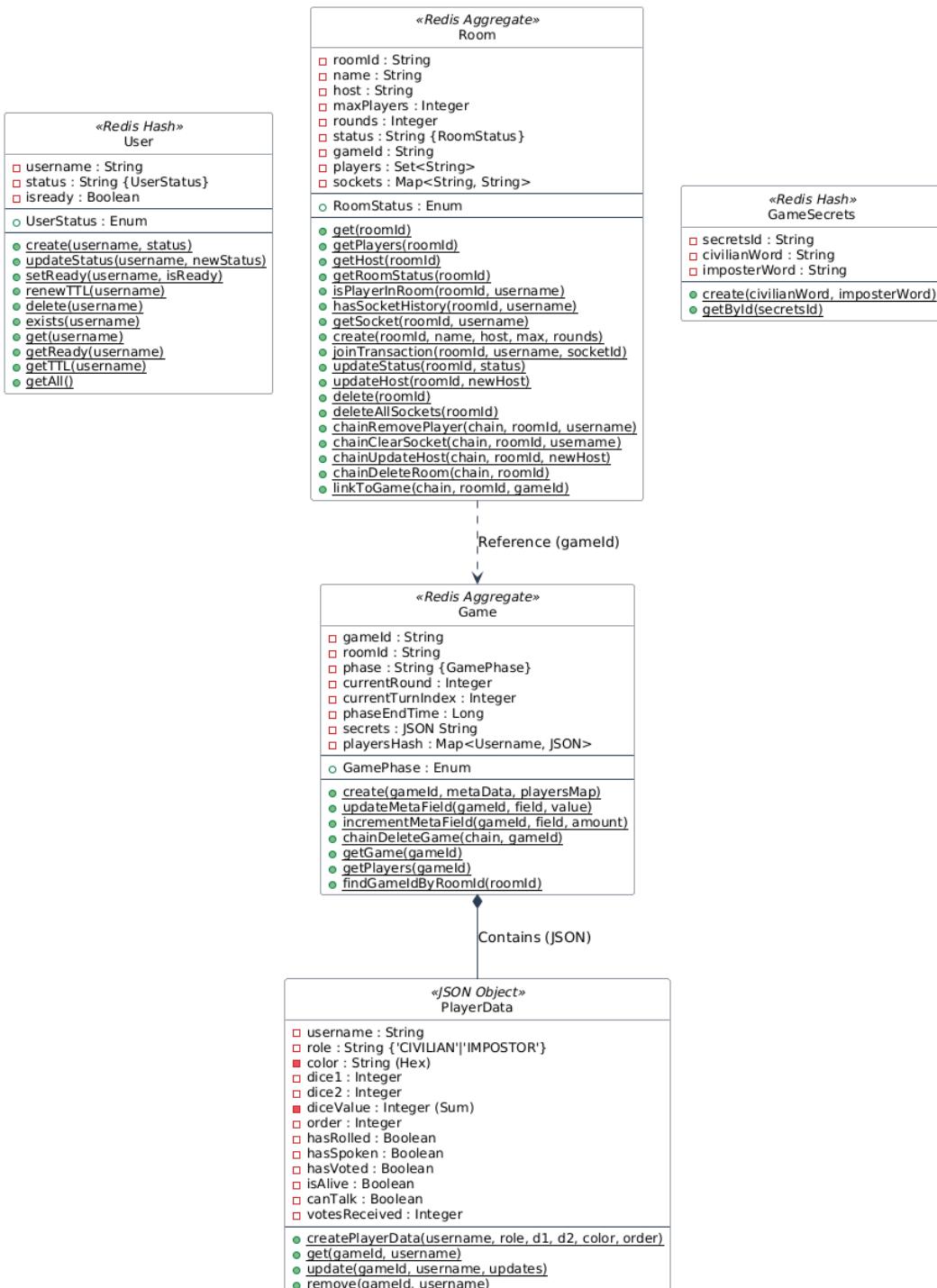
**Nota:** in realtà `lobbySockets` avrebbe dovuto interagire anch'esso con il `roomController`, ma abbiamo deciso di separare la logica degli endpoint HTTP dalla logica webSocket, per rendere più "pulita" la gestione delle **lobby**, lasciando facoltà a questo entry point di **interagire** direttamente con il **Service Layer**.

## Modellazione dei Dati

Questo diagramma rappresenta le entità logiche del sistema e le loro relazioni.

Poiché la persistenza è affidata a Redis, le classi qui rappresentate fungono da interfaccia di astrazione verso il database:

- **RedisClient**: Il modulo singleton di connessione (`redis.js`), utilizzato da tutti gli altri modelli per le operazioni di I/O (GET, SET, DEL).
- **Room & User**: Gestiscono i dati volatili della fase di Lobby e le sessioni utente.
- **Game & GameSecrets**: Gestiscono lo stato della partita.



## State Machines

### Ciclo di Vita della Room

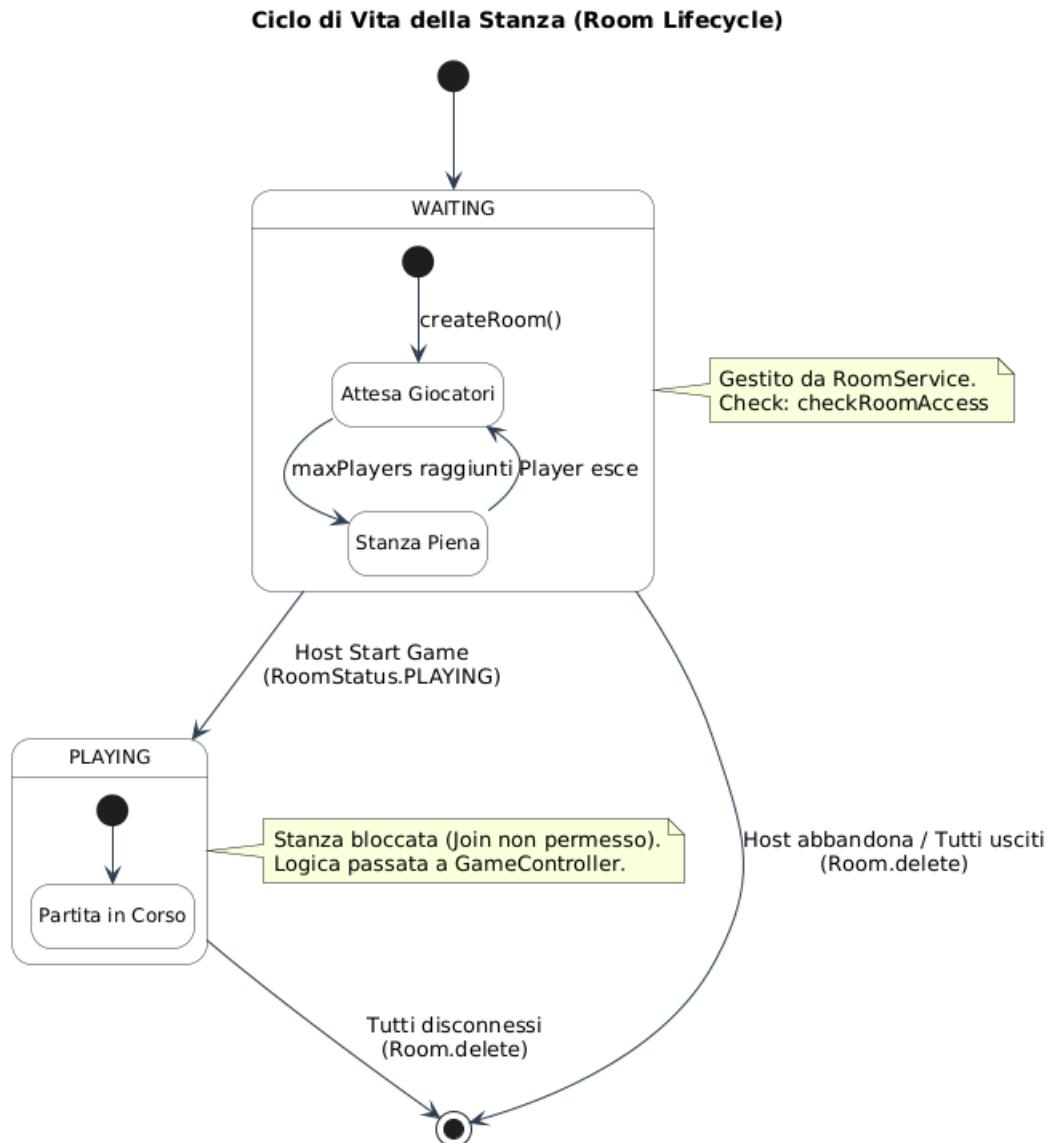
Questo diagramma illustra gli stati operativi della stanza definiti nell'enum `RoomStatus`.

La stanza nasce in stato **WAITING**, permettendo ai giocatori di unirsi fino al raggiungimento della capienza massima.

Quando l'host avvia la partita, lo stato cambia in **PLAYING**, bloccando nuovi accessi e delegando il controllo al `GameService` che creerà il **Game** e passando dunque all'interazione con il **controller**.

relativo (o meglio, passando a `gameSocket` che controllerà il `gameController`).

La room/lobby viene distrutta automaticamente quando l'ultimo giocatore la abbandona.



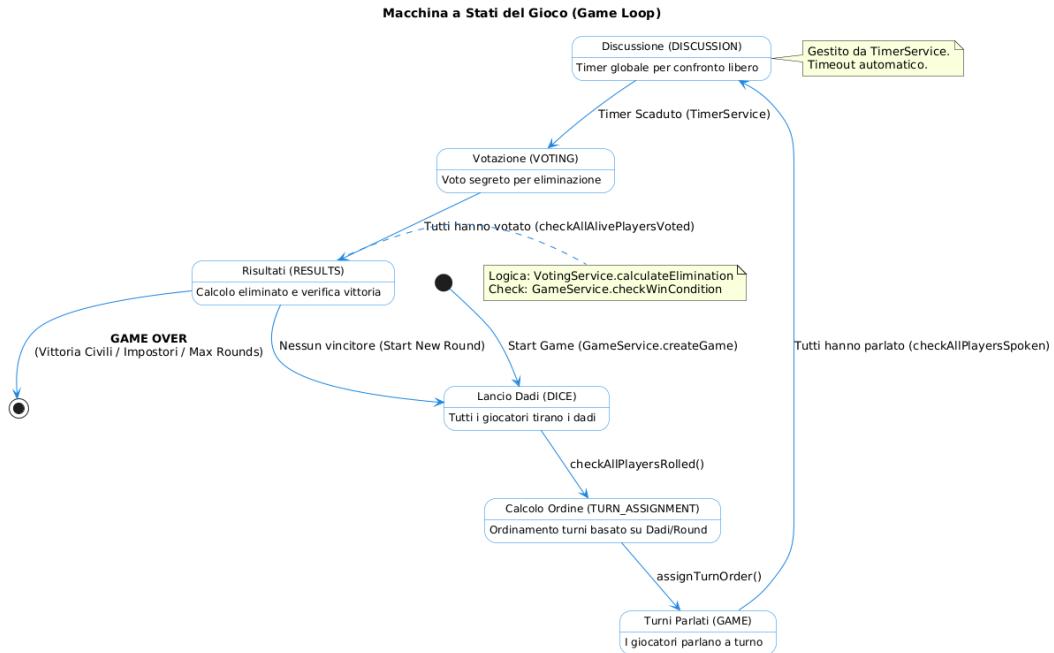
## Flusso di Gioco (Game Loop)

Questo diagramma descrive la macchina a stati finiti che governa la logica della partita, definita nell'enum `GamePhase`.

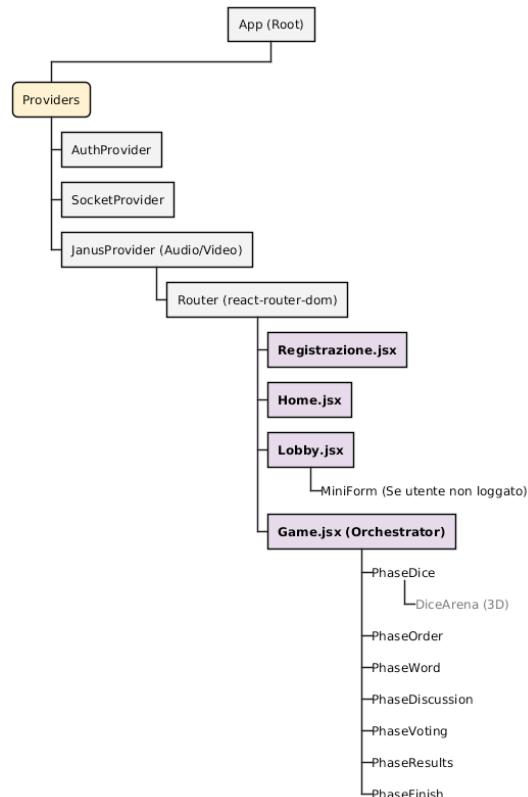
Il flusso segue un ciclo ripetitivo (Round) gestito dal `GameService`.

Il ciclo avanza in base alle azioni dei giocatori (es. lancio dadi, votazione) o allo scadere di timer gestiti dal `TimerService` (es. discussione).

Alla fine di ogni round, il sistema verifica le condizioni di vittoria (`checkWinCondition`) per decidere se terminare la partita o iniziare un nuovo round.



## Struttura Front-End



## Dettaglio delle Interazioni

# Implementazione

## Entry point Front-end: App.jsx e providers



```
1 const router = createBrowserRouter([
2   {
3     path: "/",
4     element: <Registrazione />
5   },
6   {
7     path: "/match/:roomId",
8     element: <Lobby/>
9   }
10]);
11
12 function App() {
13   return (
14     // LIVELLO 1: Gestisce "Chi sono" (User)
15     <AuthProvider>
16       {/* LIVELLO 2: Gestisce "La connessione" (Socket) */}
17       {/* Nota: SocketProvider sta DENTRO AuthProvider perché ha bisogno di 'user' */}
18       <SocketProvider>
19         {/* LIVELLO 2.5: Gestisce "La connessione Janus" */}
20         <JanusProvider>
21           {/* LIVELLO 3: Gestisce "Dove sono" (Pagine) */}
22           <RouterProvider router={router} />
23         </JanusProvider>
24       </SocketProvider>
25     </AuthProvider>
26   );
27 }
```

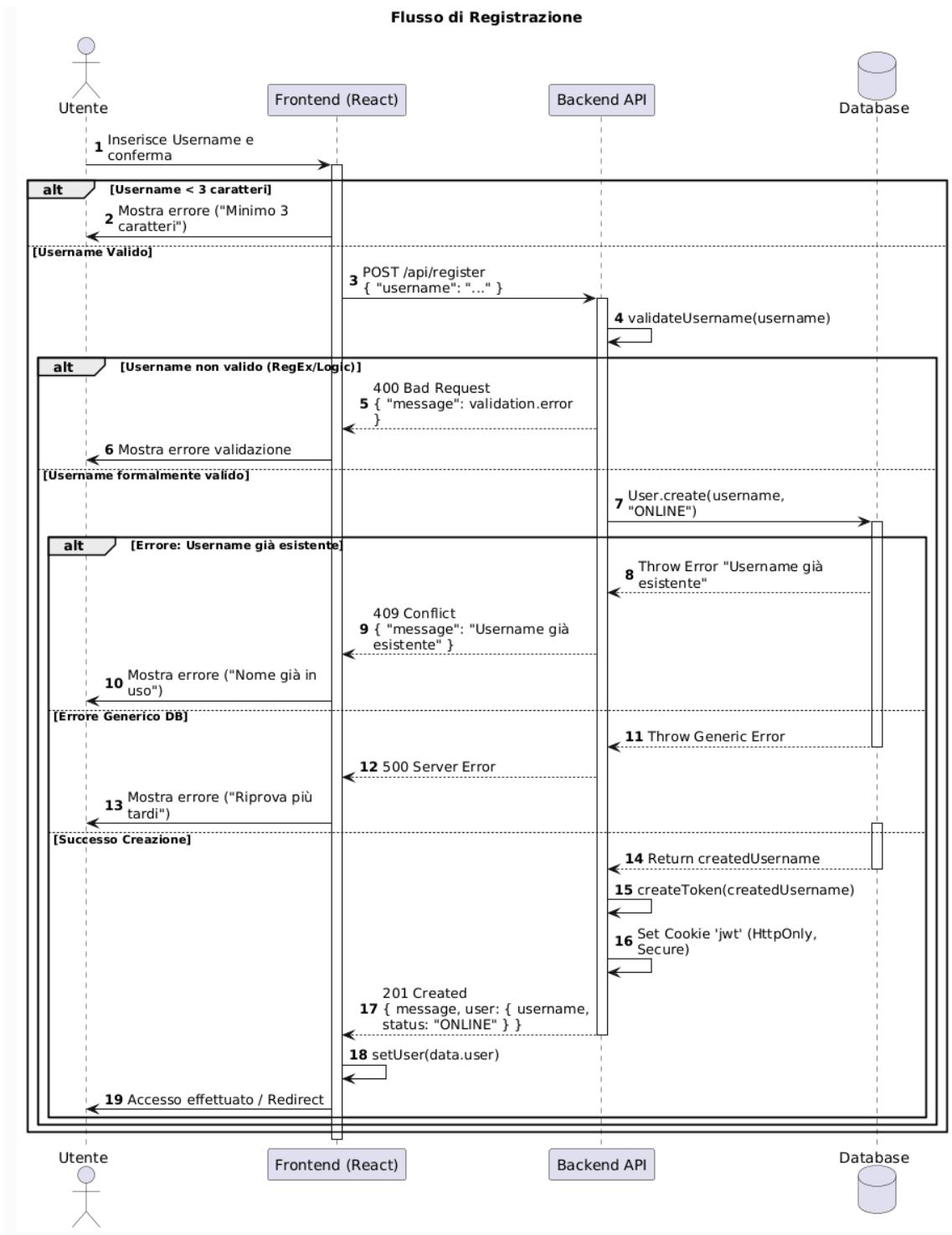
All'interno dell'entry point del nostro front-end, App.jsx abbiamo implementato 4 providers incapsulati tra loro, ognuno afferente al proprio .jsx, ad eccezione del RouterProvider implementato stesso nel componente App:

- **AuthProvider** (livello 1): recupera l'utente corrente da /api/me al mount, gestisce lo stato di caricamento e mette a disposizione user e setUser ai componenti figli, così da sapere se la sessione è attiva.
- **SocketProvider** (livello 2): mantiene una singola connessione Socket.IO condivisa. Offre metodi connectSocket per aprire il websocket una sola volta e disconnectSocket per chiuderlo e azzerare lo stato.
- **JanusProvider** (livello 2.5): orchestra l'inizializzazione della libreria Janus, la creazione della sessione/video room, la pubblicazione del flusso locale e la sottoscrizione agli stream remoti. Espone stato (es. isJanusReady, status, error, stream locali/remoti) e funzioni operative come initializeJanus, joinRoom, cleanup e toggleAudio.
- **RouterProvider** (livello 3): RouterProvider è il componente di react-router-dom che prende il router creato con `createBrowserRouter` e lo rende attivo. Si occupa di ascoltare l'URL, trovare la route giusta e renderizzare il relativo elemento fornendo inoltre il contesto di routing (navigate, params, ecc.) ai componenti discendenti.

## Registrazione



1. **Registrazione utente - handleSubmit;**



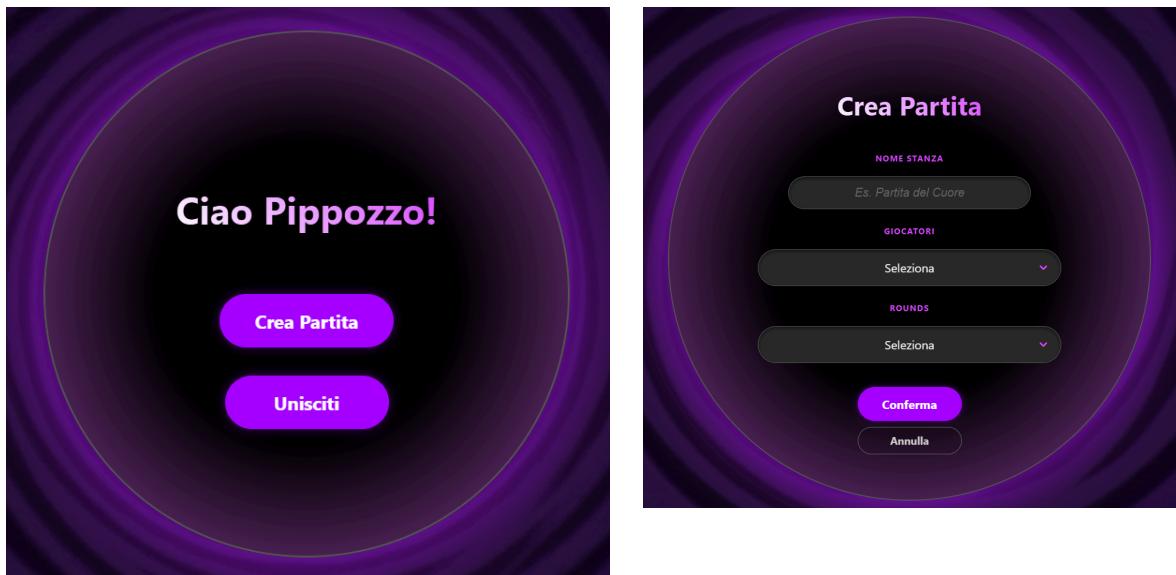
## Analisi del Flusso

Il flusso integra una validazione sia lato client che lato server, gestendo diversi scenari di errore e il successo finale con il salvataggio dello stato.

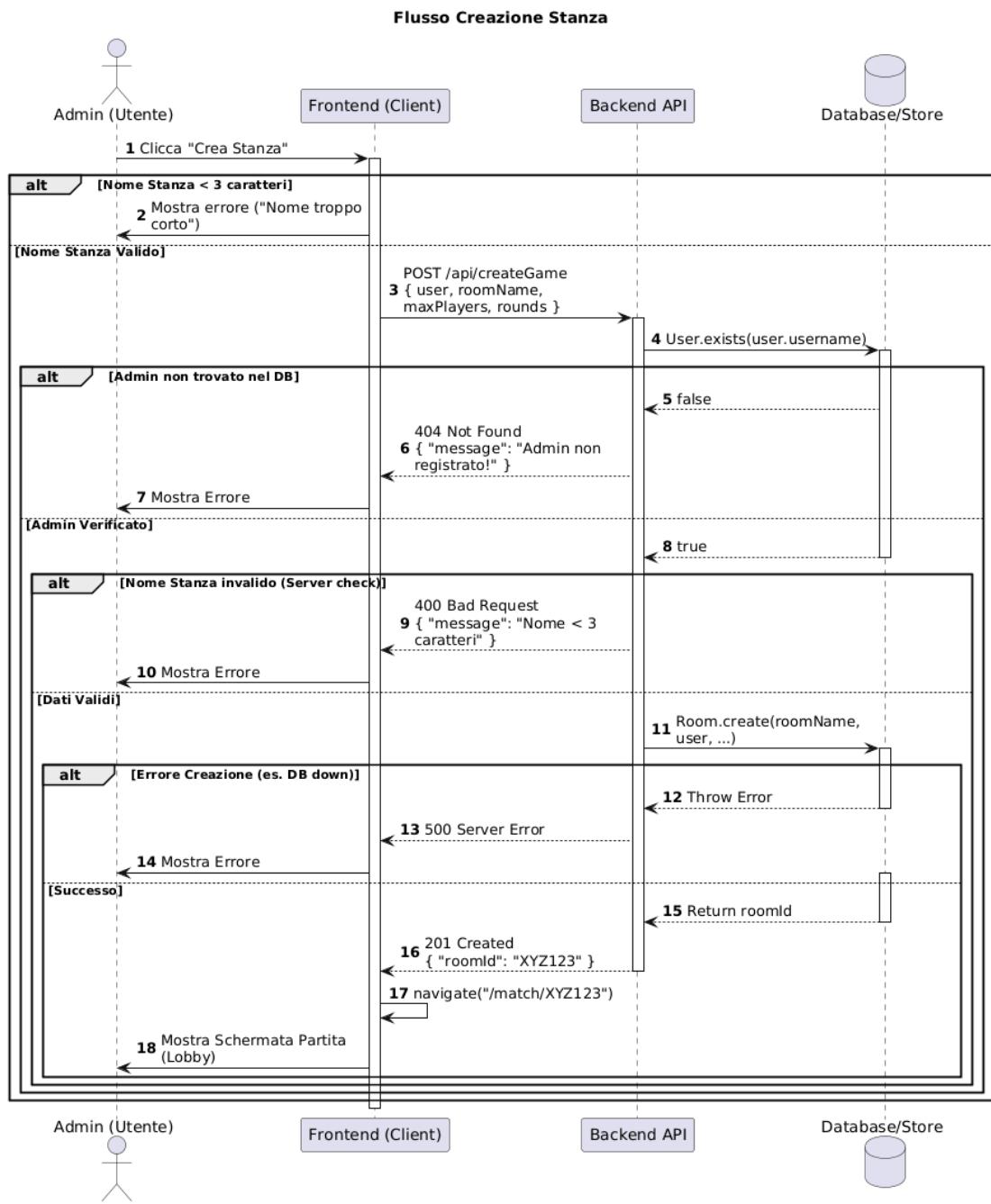
- Frontend (Pre-check):** Il frontend controlla subito se la lunghezza dell'username è inferiore a 3 caratteri per evitare chiamate API inutili.
- Richiesta API:** Se il controllo passa, viene inviata una `POST` a `/api/register`. Notare `credentials: 'include'`, fondamentale per permettere al browser di ricevere e salvare il cookie inviato dal server.

- **Backend (Validazione):** Il server riceve la richiesta e usa una funzione `validateUsername`. Se fallisce, risponde con **400**.
- **Backend (Creazione):** Tenta di creare l'utente nel DB (`User.create`).
  - Se l'utente esiste già (gestito nel `catch`), risponde con **409**.
  - Se c'è un errore generico, risponde con **500**.
- **Backend (Successo):** Se tutto va bene, genera un JWT, lo imposta come cookie `HTTPOnly` e restituisce l'oggetto utente con stato **201**.
- **Frontend (Conclusione):** Il frontend riceve la risposta, aggiorna lo stato globale (`setUser`) e l'interfaccia reagisce di conseguenza.

## Home



### 1. Creazione stanza



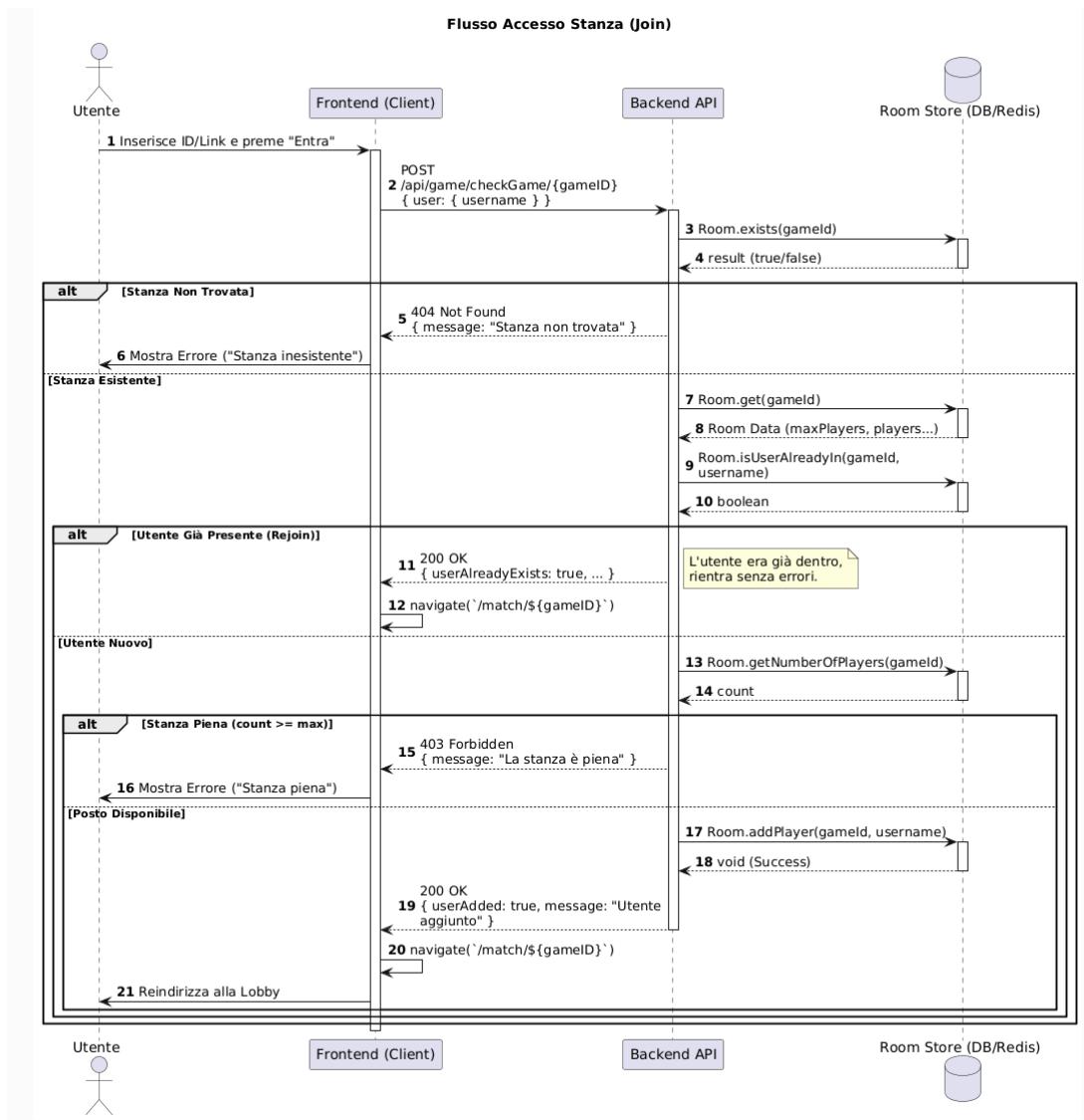
### Analisi del Flusso

Questa funzionalità è più articolata della registrazione perché coinvolge un controllo incrociato: prima di creare la risorsa (Stanza), il backend verifica che chi sta facendo la richiesta (Admin) esista ancora nel sistema.

- **Frontend (Validazione Pre-invio):**
  - L'utente clicca su "Crea Stanza".
  - Controllo immediato: se il nome della lobby è inferiore a 3 caratteri, mostra errore e ferma tutto.
- **Richiesta API:**

- Invia una `POST` a `/api/createGame` includendo nel body l'oggetto `user`, `roomName`, `maxPlayers` e `rounds`.
- **Backend (Verifica Integrità Utente):**
  - Prima di validare i dati della stanza, il server interroga il Database (`User.exists`) per assicurarsi che l'utente ("Admin") sia registrato.
  - Se l'utente non esiste (es. sessione scaduta o utente cancellato), restituisce **404**.
- **Backend (Validazione Dati Stanza):**
  - Controlla la lunghezza del nome della stanza. Se invalido, restituisce **400**.
- **Backend (Creazione Risorsa):**
  - Chiama `Room.create(...)` per istanziare la partita (probabilmente su DB o Redis).
  - Restituisce **201** con il `roomId` appena generato.
- **Frontend (Navigazione):**
  - Riceve l'ID della stanza e forza il reindirizzamento (`Maps`) alla rotta della partita (`/match/:id`).

## 2. Unisciti alla stanza - handleJoinGame;



## Analisi del Flusso

Questa è una funzione critica perché gestisce la concorrenza (stanza piena) e lo stato (utente già presente). Nota interessante: nonostante il nome della funzione backend sia `checkGameP` e il commento frontend dica "vedere solo se la partita esiste", il codice in realtà **modifica lo stato** aggiungendo il giocatore (`Room.addPlayer`) se le condizioni sono soddisfatte.

- **Frontend (Richiesta):**

- Invia una `POST` (nota: nel commento c'è scritto GET, ma il codice esegue una POST) a `/api/game/checkGame/{id}`.

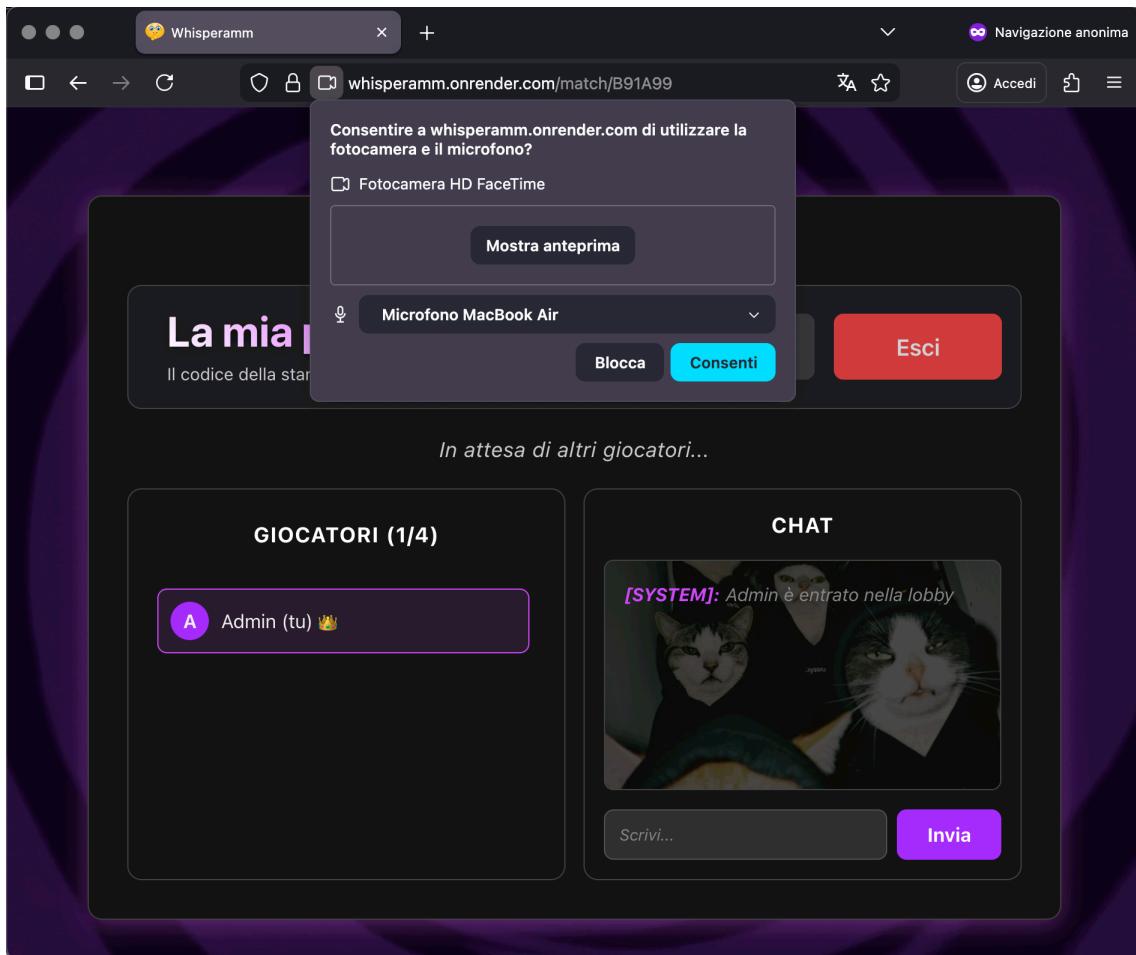
- **Backend (Validazione a cascata):**

- Check Dati:** Se manca l'oggetto `user` nel body, errore **400**.
- Check Esistenza Stanza:** Se l'ID non esiste, errore **404**.
- Check "Già Presente" (Rejoin):** Se l'utente è già nella lista giocatori, il server risponde **200 OK** (ma con flag `userAlreadyExists: true`). Questo è ottimo per gestire refresh di pagina o riconnessioni.
- Check Capienza:** Se l'utente *non* è dentro e `giocatori >= max`, errore **403 (Forbidden)**.

- 5. Azione di Scrittura:** Se c'è posto, chiama `Room.addPlayer` per inserire l'utente.
- **Backend (Risposta):** Restituisce **200 OK** con flag `userAdded: true`.
  - **Frontend (Reazione):** Se lo status è 200 (sia per nuovo accesso che per rejoin), esegue la `Maps` verso la rotta del match.

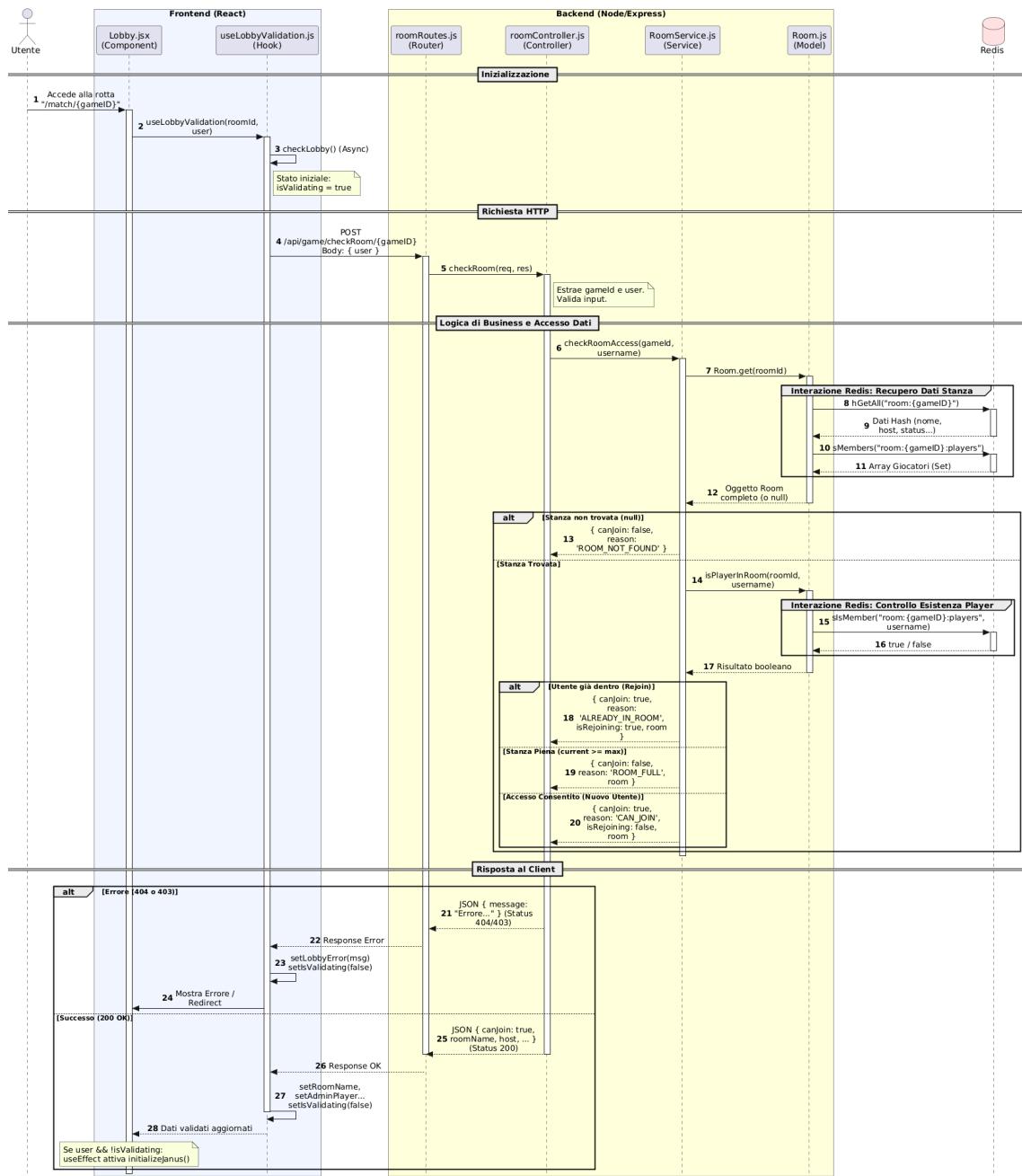
## Lobby

Una volta che l'admin ha creato la partita verrà quindi effettuato il render di `Lobby.jsx`.



Innanzitutto, all'accesso in lobby viene effettuato un controllo di esistenza e vincoli (numero giocatori massimo) sulla stanza oltre che l'aggiunta del Player alla lobby nel backend, questo è necessario in quanto è consentito agli utenti di accedere tramite codice univoco e link.

### 1. Controllo sulla stanza - `checkLobby`;



## Analisi del Flusso

- Avvio (Frontend):** L'utente richiede la rotta `/match/{gameID}`.  
Il componente `Lobby.jsx` monta ed esegue l'hook `useLobbyValidation`. La UI viene bloccata in stato di caricamento ("Verifica...").
- Richiesta di Verifica (HTTP):** il client invia una richiesta **POST** all'endpoint `/api/game/checkRoom/{gameID}`, allegando l'oggetto `user` nel body.
- Interrogazione Redis (Backend - Read Only):** il backend (`RoomService`) interroga Redis tramite il Model per verificare lo stato della stanza:
  - Esistenza:** Recupera i metadati della stanza (`hGetAll`). Se null, restituisce **404**.
  - Identità Utente:** Controlla se l'utente è già nel set dei giocatori (`sIsMember`). Se sì, segnala che è un **Rejoin**.

- **Capienza:** Se non è un rejoin, conta i membri attuali ( `sMembers` ) e li confronta con `maxPlayers` . Se piena, restituisce **403**.

4. **Risposta (Backend → Frontend):** il server risponde con un JSON:

- **Caso OK (200):** Restituisce i dati della stanza (nome, host, impostazioni) e il flag `canJoin: true` .
- **Caso Errore:** Restituisce il messaggio di errore specifico (Stanza non trovata / Stanza piena).

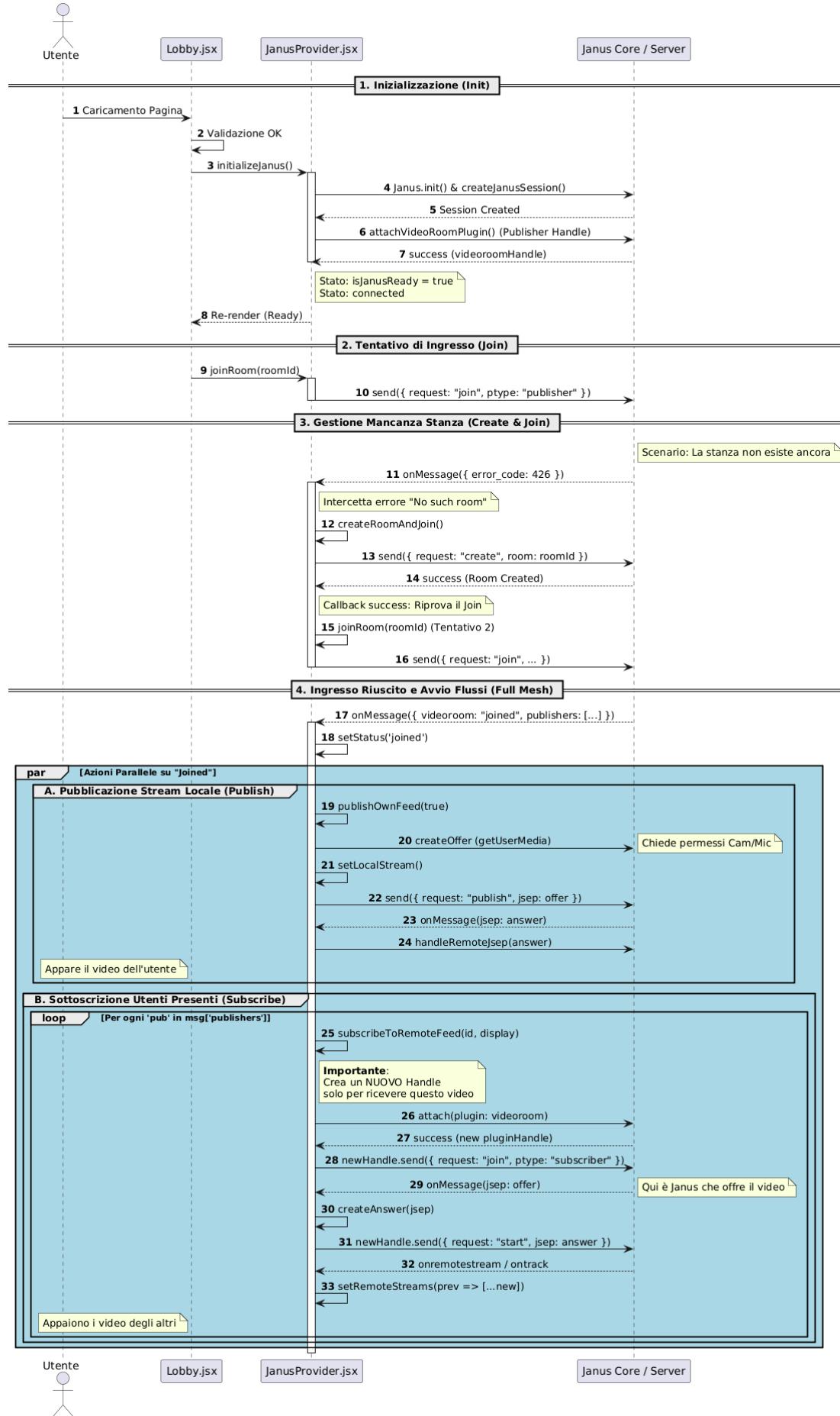
5. **Aggiornamento UI (Frontend)**

- **Se Successo:** L'hook aggiorna gli stati locali ( `roomName` , `isAdmin` , ecc.) e disattiva il caricamento. Questo sblocca innesca a cascata l'inizializzazione di **Janus** (video) e la connessione **Socket** (chat/gioco).
- **Se Errore:** Viene mostrato un messaggio di errore all'utente e dopo 2 secondi avviene un redirect alla Home.

---

In lobby viene soprattutto avviata la logica di trasmissione vocale e video del player, in particolare viene chiesta l'autorizzazione per l'accesso ai dispositivi di trasmissione e gestita la logica secondo cui la concessione di questa è un requisito obbligatorio per la partecipazione al gioco.

2. **Interazione con Janus Media Server:**



## Analisi del flusso

1. **Fase Iniziale: Setup e Aggancio al Server (Inizializzazione):** la `Lobby` funge da innesco.  
Appena l'utente è validato, il componente ordina al `JanusProvider` di partire (`initializeJanus`).  
Il Provider lavora "dietro le quinte" per stabilire l'infrastruttura di base:
  - Inizializza la libreria (`Janus.init`).
  - Crea la sessione HTTP con il server (`createJanusSession`).
  - Effettua il primo **Attach** al plugin `VideoRoom`. Questo crea il "**Publisher Handle**", ovvero il canale principale che servirà per inviare comandi e trasmettere il proprio video.
2. **Fase reattiva:** una volta che il plugin è agganciato, il Provider alza la flag `isJanusReady`.  
La `Lobby`, che monitora costantemente questa variabile tramite un `useEffect`, reagisce immediatamente inviando il comando logico di ingresso: `joinRoom(roomId)`.  
*Nota:* Qui non stiamo ancora trasmettendo video, stiamo solo chiedendo il permesso di entrare nella stanza logica.
3. **Fase di Resilienza: Intercettazione e Creazione Stanza (Auto-Recovery):** se il comando `join` fallisce perché la stanza non esiste (il server risponde con **errore 426** in `onJanusMessage`), il Provider non mostra l'errore all'utente ma agisce in autonomia:
  1. Intercetta l'errore.
  2. Chiama internamente `createRoomAndJoin`.
  3. Invia una richiesta `create` al server.
  4. Al successo della creazione, ritenta automaticamente il `joinRoom`.
4. **Fase Esecutiva: Negoziazione (Publish & Subscribe):** quando il server conferma l'ingresso con l'evento `joined`, il Provider scatena due processi paralleli fondamentali:
  - **A. Pubblicazione Stream Locale (Publish):**  
Chiama `publishOwnFeed(true)`. Il browser chiede i permessi per cam/microfono (`getUserMedia`), crea l'Offerta WebRTC (SDP) e la invia al server tramite l'Handle principale.
  - **B. Sottoscrizione Stream Remoti (Subscribe):**  
Se nell'evento `joined` sono elencati altri utenti (`publishers`), il Provider itera su di essi e per ognuno chiama `subscribeToRemoteFeed`.
    - *Dettaglio Tecnico:* Per ogni utente remoto da vedere, Janus crea un **nuovo Handle separato** (una nuova `attach` al plugin).
    - Il Provider negozia una connessione in ricezione (Subscriber) su questo nuovo handle.

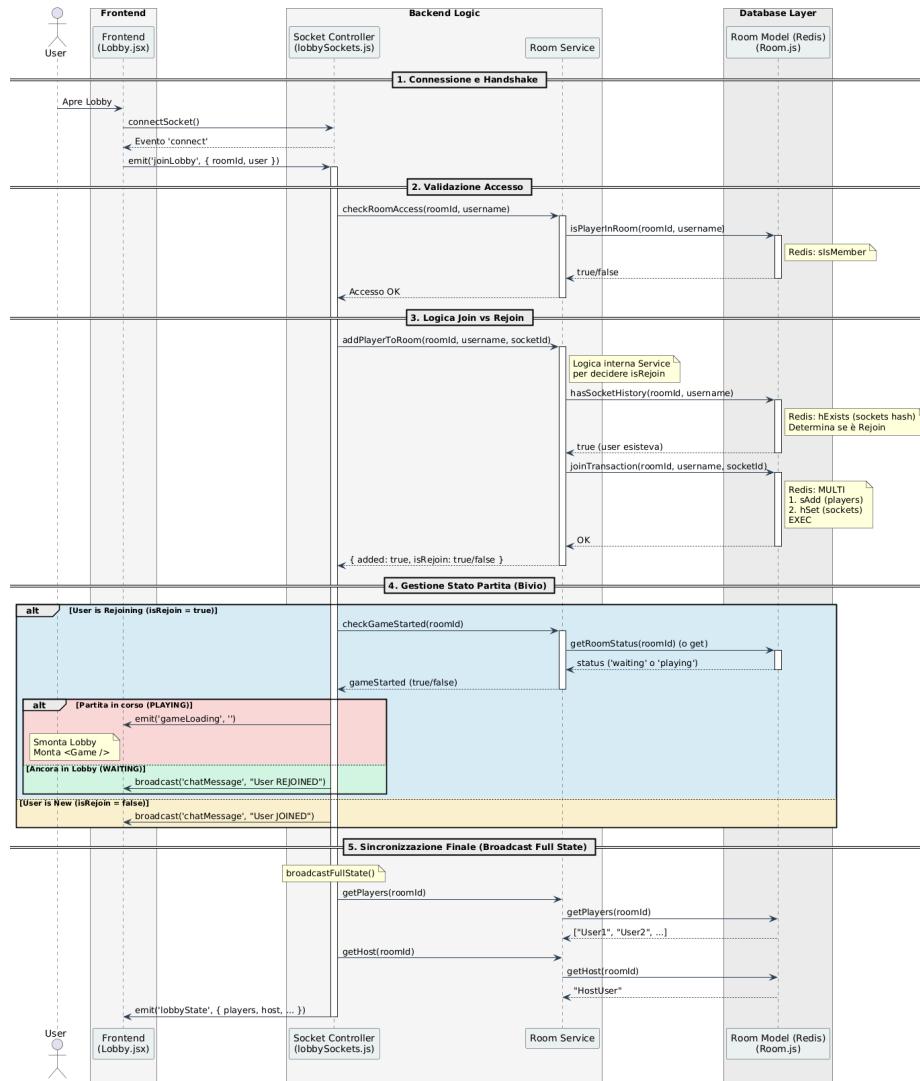
Nota: la comunicazione tra Client-Janus Media Server, e il conseguente plugin VideoRoom avviene sfruttando HTTP Long Poll, e non webSocket.

Viene poi eseguito la connessione della socket verso il server Node e la gestione di tutte le operazioni associate:

- Gestione dei pronto;
- Chat;
- Notifiche;

- Disconnessione del player;

### 3. Connessione socket - socketConnect;



### Analisi del Flusso

Il sistema è progettato per essere **resiliente alle disconnessioni**: abbiamo fatto in modo che il Backend non si fidi solo della connessione attuale, ma controlla la "storia" dell'utente su Redis per decidere se trattarlo come nuovo o come utente di ritorno.

#### 1. Innesco (Frontend):

- **Azione:** L'utente atterra in Lobby (`Lobby.jsx`).
- **Hook:** `useLobbySocket` stabilisce la connessione Socket.IO.
- **Event:** Appena connesso, emette `joinLobby` con `{ roomid, user }`.

#### 2. Elaborazione & Persistenza (Backend):

il controller riceve l'evento e interroga il Model `Room.js`:

- **Check Esistenza:** Verifica se l'utente esiste già nella mappa `room:{id}:sockets` (metodo `hasSocketHistory`).
  - **Se esiste:** Flag `isRejoin = true`.

- Se non esiste: Flag `isRejoin = false`.
- **Scrittura Atomica:** Esegue una transazione Redis (`joinTransaction`) che fa due cose simultaneamente:
  1. Aggiunge l'utente al Set dei giocatori (`players`).
  2. Aggiorna l'Hash socket (`sockets`) con il **nuovo ID socket**.
- 3. **Biforcazione Decisionale:** sulla base del flag `isRejoin` e dello stato della stanza (`RoomStatus`), il server decide il destino dell'utente:
  - **CASO A: Nuovo Utente ( `isRejoin: false` )**
    - L'utente viene aggiunto alla Lobby.
    - Broadcast messaggio: "*Username è entrato*".
  - **CASO B: Rejoin in Lobby ( `isRejoin: true` && Status: `WAITING` )**
    - L'utente aveva perso la connessione ma la partita non è iniziata.
    - Resta in Lobby.
    - Broadcast messaggio: "*Username si è riconnesso*".
  - **CASO C: Rejoin in Game ( `isRejoin: true` && Status: `PLAYING` )**
    - L'utente era in partita ed è caduto/ha fatto refresh.
    - Il server invia l'evento `gameLoading` **solo a lui**.
    - **Risultato UI:** Il Frontend smonta la `Lobby` e monta il componente `Game`, permettendo all'utente di riprendere a giocare immediatamente.

#### 4. Sincronizzazione Finale (per consistenza)

Indipendentemente dal caso, viene invocata `broadcastFullState`:

- Il server scarica da Redis la "fotografia" aggiornata della stanza (lista player, chi è pronto, chi è l'host).
- Invia `lobbyState` a **tutti** i client connessi per allineare le interfacce grafiche.

---

I vantaggi di quest'implementazione sono:

1. **Resistenza al F5:** Se l'utente ricarica la pagina, Redis si ricorda di lui (`hasSocketHistory`), il Socket ID viene aggiornato, e l'utente viene reindirizzato correttamente (in Lobby o in Game) senza perdere il posto.
  2. **Transazioni Atomiche:** L'uso di `client.multi()` in `Room.js` garantisce che non ci siano mai giocatori "a metà" (presenti nella lista ma senza socket, o viceversa).
  3. **Separazione UI:** Il Frontend è stato reso stupido; non decide se mostrare il Gioco o la Lobby al caricamento. È il Backend che gli "ordina" di caricare il gioco (`gameLoading`) se necessario.
- 

## Game

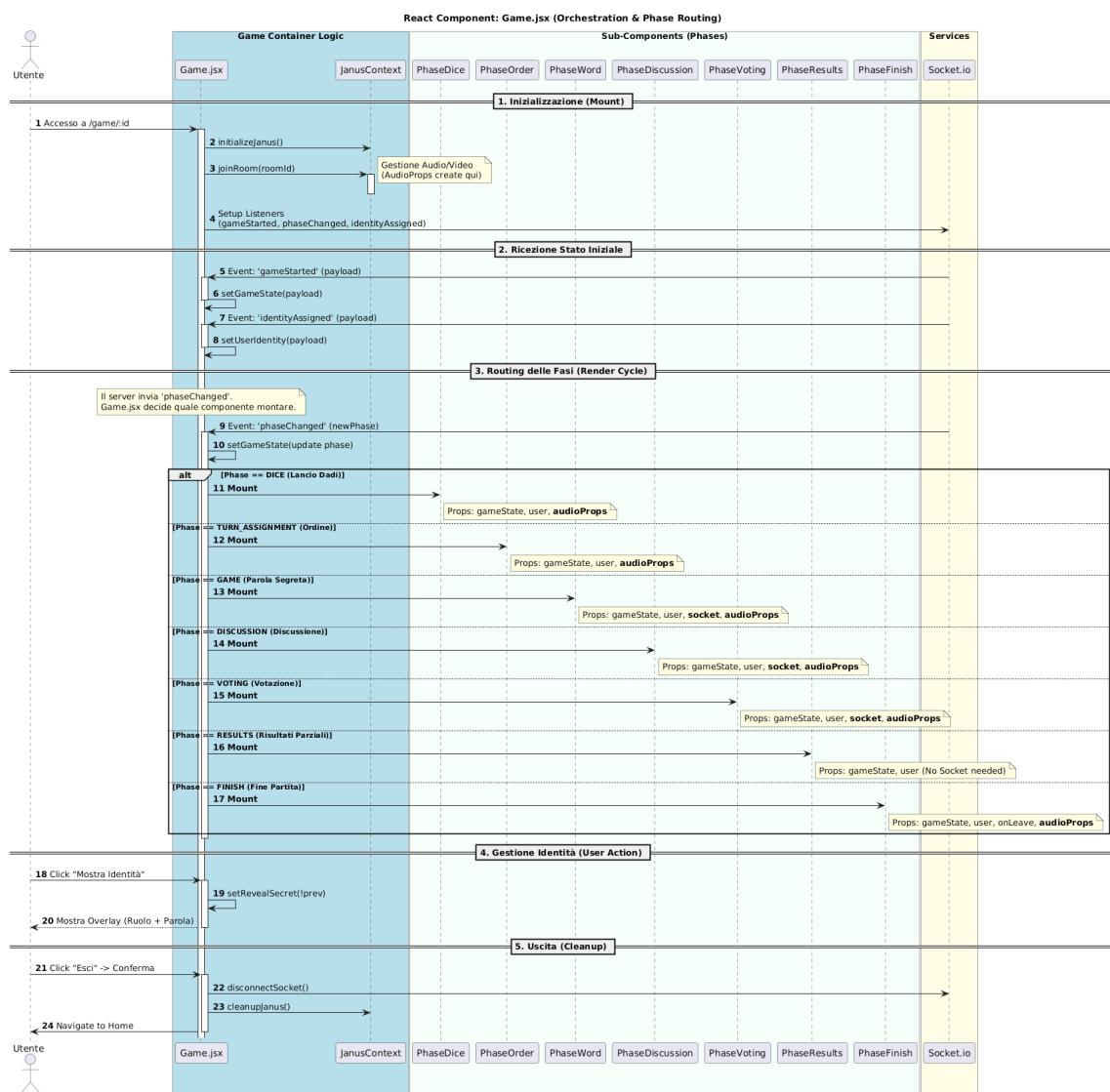
Una volta che tutti gli utenti sono pronti avviene una **Renderizzazione condizionale** secondo i principi della Single Page Application.

```
if (gameLoading) return <Game />;
```

È importante notare che, poiché `Game` viene renderizzato dentro `Lobby`, il contesto non viene perso:

- La connessione Socket rimane attiva.
- L'utente (`user`) rimane autenticato.
- La connessione Janus (video/audio) rimane attiva.

Procediamo ora ad un'analisi del componente React Game che come vedremo agirà da **orchestratore**, in quanto **terrà traccia delle connessioni globali** (WebSocket e Janus) oltre a **Mantenere lo stato del gioco** per prendere decisioni rispetto a quale sotto-componente visualizzare. La cosa importante è che la logica di ogni fase sarà quindi delegata a ognuno dei sotto-componenti.



Dal diagramma di sequenza siamo in grado di visualizzare in maniera rapida anche quali delle fasi di gioco richiedono la gestione tramite eventi **webSocket** e quali no.

Mentre per quanto riguarda l'oggetto **audioProps** questo viene creato in Game e passato in cascata, in quanto in tutte le fasi avremo bisogno di mantere il flusso audio/video attivo tra i giocatori.

Per continuare con l'analisi logica del gioco ci concentreremo sulle due fasi in cui la logica e la comunicazione tra front-end e back-end diventa più interessante.

## phaseWord

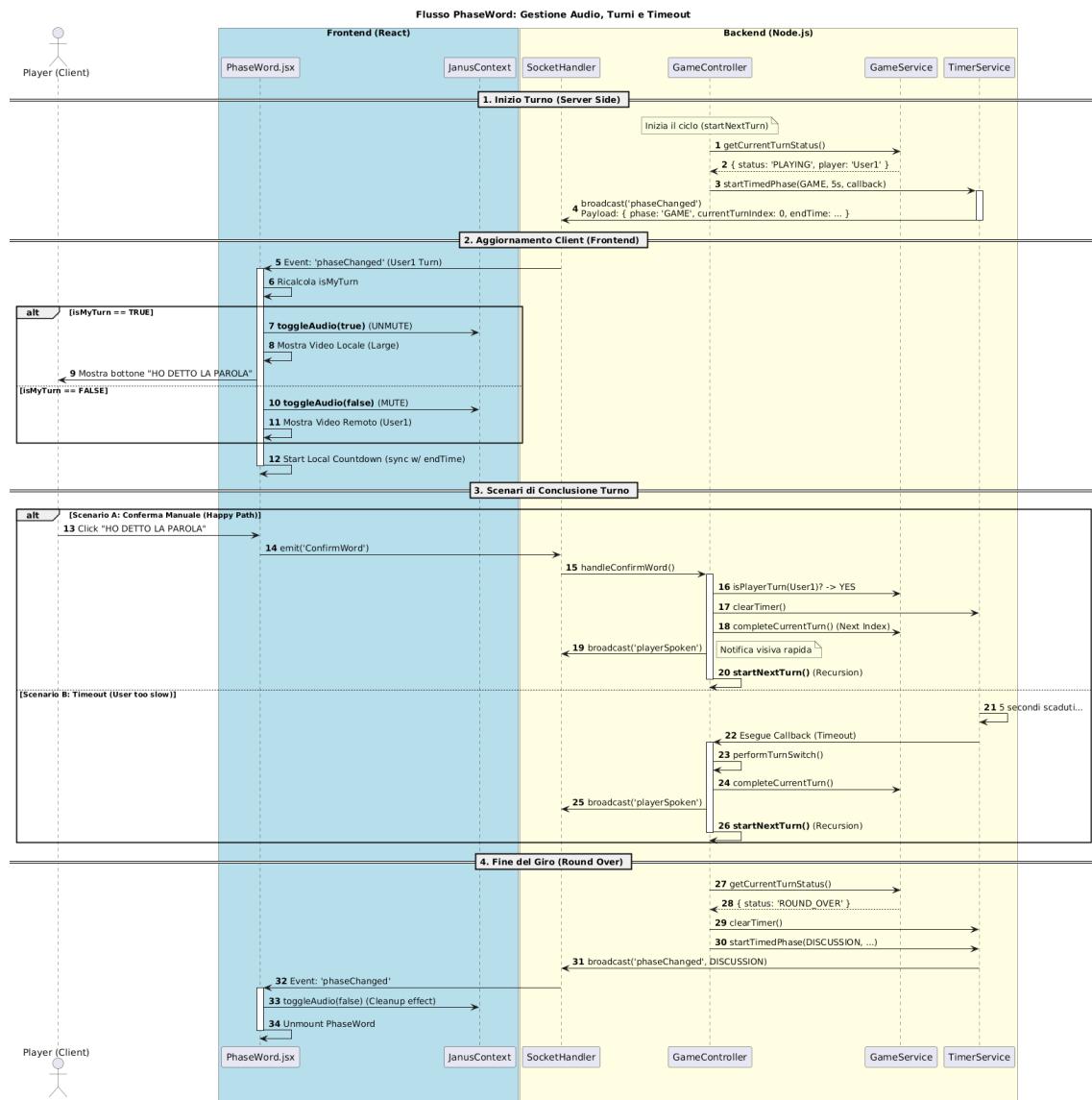
La fase `PhaseWord` (o "GAME") è il momento in cui i giocatori, a turno, devono pronunciare la loro parola segreta.

### Logica Frontend:

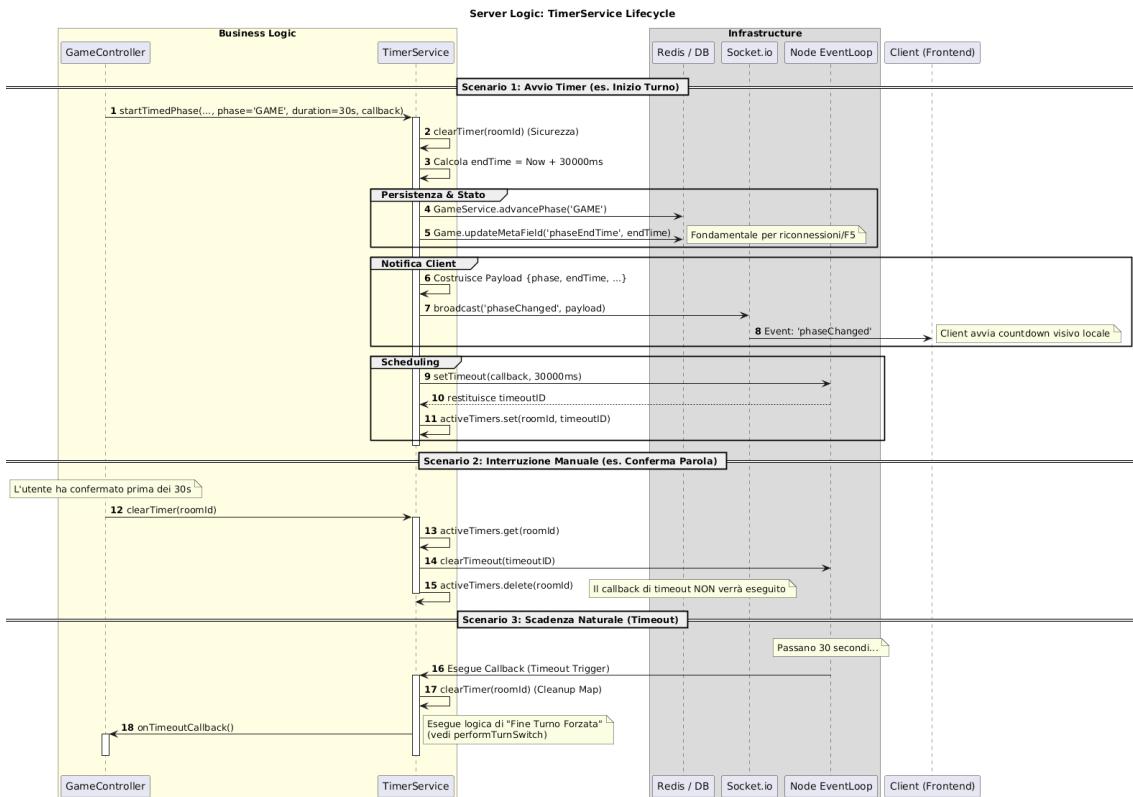
Come anticipato andiamo a effettuare la gestione dell'audio "intelligente" direttamente sul front-end. L'hook `useEffect` dipendente da `isMyTurn` garantisce che solo il giocatore attivo trasmetta audio, prevenendo interruzioni.

### Logica Backend:

- Il metodo `startNextTurn` determina chi deve giocare. Se il round è finito, passa alla fase `DISCUSSION`.
- Il turno avanza in due modi:
  1. **Manuale:** L'utente invia `ConfirmWord`.
  2. **Automatico (Timeout):** Il `TimerService` scatta e forza il passaggio ( `performTurnSwitch` ).



Il **TimerService** agisce come **Unica Fonte di Verità (Single Source of Truth)** per il tempo di gioco. Disaccoppia la gestione del tempo dalla logica di gioco, offrendo un'interfaccia pulita per avviare fasi a tempo.



### Le funzionalità chiave:

- Utilizza una mappa in memoria (`roomId` → `timeoutId`) per tracciare i timer attivi. Questo permette di cancellare bruscamente un timer se un evento di gioco avviene prima della scadenza.
- Calcola il timestamp assoluto di fine (`endTime`) e lo salva su Redis (`phaseEndTime`). Questo è cruciale: se un utente ricarica la pagina (F5), il frontend può leggere dal DB quanto tempo manca realmente, senza perdere la sincronizzazione.
- Notifica immediatamente a tutti i client il cambio di fase, includendo `endTime`, permettendo ai frontend di avviare le loro animazioni di countdown in modo sincronizzato.
- Accetta una funzione (`onTimeoutCallback`) che viene eseguita solo se il tempo scade naturalmente, permettendo al Controller di definire cosa succede "dopo".

Conclusa la fase centrale di gioco, seguita da una fase di discussione procediamo con la fase di votazione, di seguito l'analisi.

### phaseVoting

La fase `PhaseVoting` (o "VOTING") presenta una griglia visiva di tutti i partecipanti. È il momento del giudizio, dove i giocatori devono decidere chi eliminare o se astenersi.

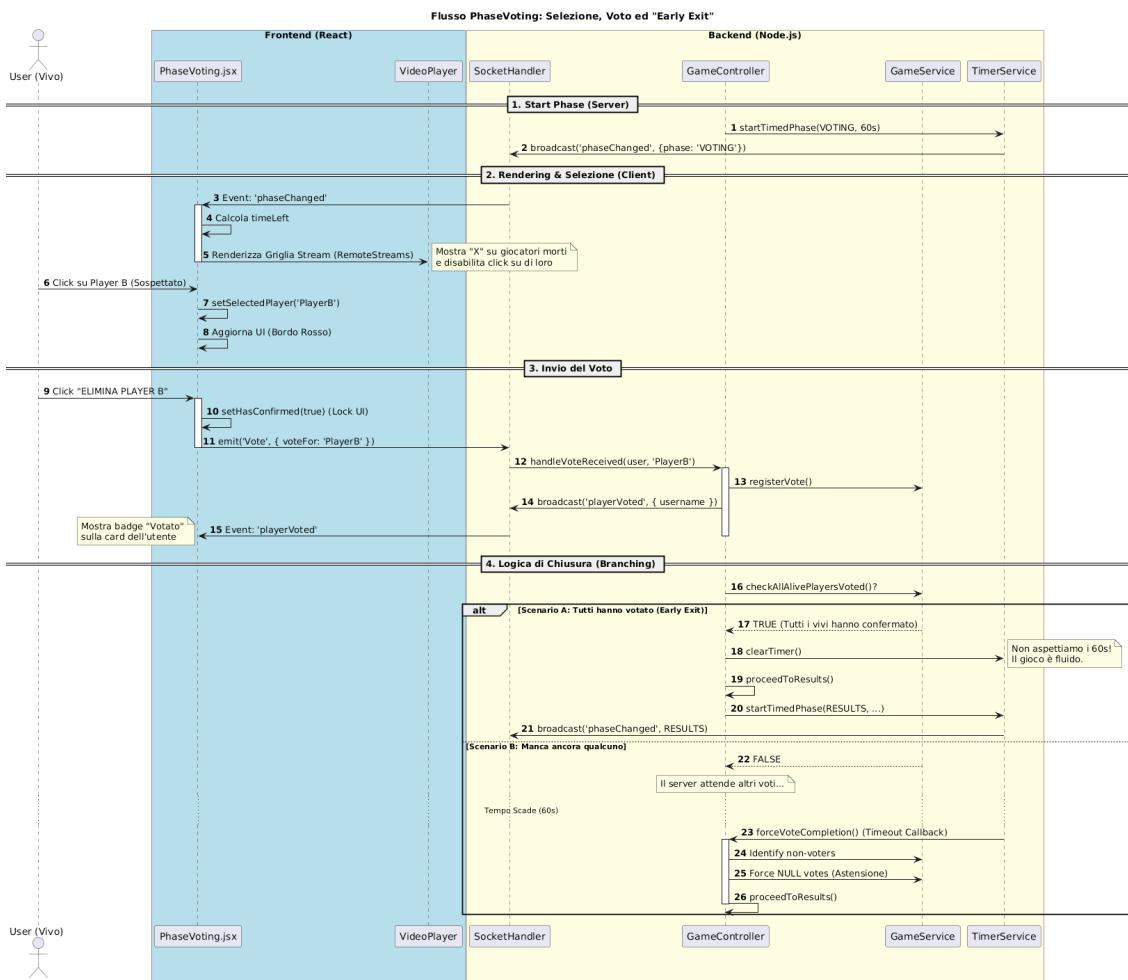
#### Logica Frontend:

- Renderizza tutti i giocatori (vivi e morti). I vivi hanno il loro stream video, i morti hanno un indicatore visivo ("X" e bordo grigio) e sono disabilitati al click.
- Logica di Selezione:**
  - L'utente può selezionare un altro giocatore cliccando sulla sua card.
  - Il tasto di conferma ("ELIMINA") si attiva solo dopo la selezione.

- L'astensione è sempre disponibile tramite un tasto dedicato.
- Una volta inviato il voto, l'interfaccia si blocca per impedire modifiche, mostrando lo stato "In attesa degli altri".

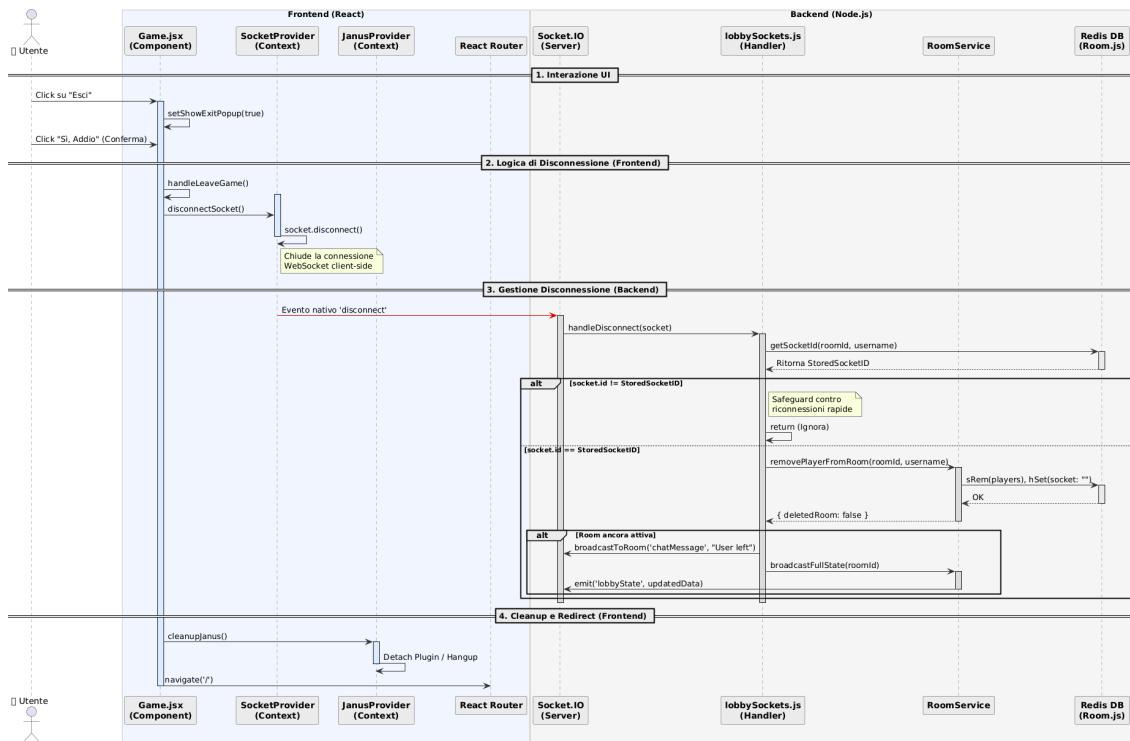
### Logica Backend:

- **Early Exit:** Questa è la *feature* più importante. Il server controlla ad ogni voto se *tutti* i giocatori vivi hanno votato. In caso affermativo, cancella il timer e passa immediatamente ai risultati, senza aspettare i 60 secondi.
- **Voto Nullo Forzato:** Se il timer scade, il server forza un voto nullo (astensione) per chi non ha fatto in tempo, garantendo che il calcolo dei risultati abbia sempre un set di dati completo.



Nel componente Game c'è un pulsante per uscire dalla partita, la cui pressione oltre a redirezionare l'utente nella home, pulirà anche la connessione websocket instaurata precedentemente, disconnettendo l'utente e pulirà la sessione Janus pregressa:

### handleLeaveGame:



## Analisi del flusso

### 1. Frontend (Game.jsx):

- L'utente clicca il pulsante per uscire.
- Viene chiamata `handleLeaveGame`.
- La funzione chiave è `disconnectSocket()` (dal hook `useSocket`). Questo taglia fisicamente la connessione WebSocket lato client.

### 2. Rete:

- Il server riceve l'evento nativo `disconnect`. Non è un evento emesso manualmente (`emit`), ma l'evento standard di Socket.IO quando cade la linea.

### 3. Backend (lobbysocket.js):

- `handleDisconnect` intercetta l'evento.
- Check di Sicurezza**: Verifica su Redis (`SocketService.getSocketId`) se l'ID del socket che si sta disconnettendo corrisponde a quello salvato per quell'utente. Questo previene bug in caso di riconnessioni rapide (F5).
- Rimozione**: Chiama il service per rimuovere l'utente dal set dei giocatori su Redis.
- Notifica**: Se la stanza è ancora attiva, avvisa gli altri client ("User ha lasciato la lobby") e invia il nuovo stato (lista giocatori aggiornata) tramite `broadcastFullState`.

### 4. Frontend (Cleanup finale):

- Dopo aver staccato il socket, `Game.jsx` pulisce anche la sessione audio/video (`cleanupJanus`) e reinDIRIZZA l'utente alla Home (`Maps('/')`).

## Janus Media Server

Per implementare un Media Server Janus abbiamo optato per l'utilizzo di una VM nel cloud tramite OCI (Oracle Cloud Infrastructure), il motivo di questa scelta è stato guidato allo scopo di agevolarci nella fase di deploy che è seguita al completamento dello sviluppo dell'applicazione.

- All'interno della VM abbiamo installato Janus seguendo le istruzioni presenti in <https://github.com/meetecho/janus-gateway>
- Per la configurazione del server abbiamo modificato i seguenti file:
  1. janus.jcfg, presente nella directory "/opt/Janus/etc/janus"
    - a. abbiamo aggiunto il server stun di google nella configurazione del nat

```
stun_server = "stun.l.google.com"  
stun_port = 19302
```

- b. il nat\_mapping 1\_1 specificando l'indirizzo IP pubblico della VM

- Sono state modificate inoltre le regole di firewalling della VM per permettere il corretto funzionamento del Media Server
  - Esposizione dell'endpoint per l'API RESTful Janus

0.0.0.0/0	TCP	All	8088	TCP traffic for ports: 8088
-----------	-----	-----	------	-----------------------------

- Esposizione delle porte usate da RTP per le sessioni multimediali

0.0.0.0/0	UDP	All	10000-60000	UDP traffic for ports: 10000-60000
-----------	-----	-----	-------------	------------------------------------

## Deploy

Nella prima macrofase del ciclo di vita dell'applicazione, ovvero il Development, abbiamo fatto girare il front-end e il back-end su due porte separate (server Vite e node dev), facendone il binding per il testing dell'applicazione nelle prime fasi.

Nella fase successiva, quella di Production, abbiamo:

- Ottimizzato l'ambiente **buildando** il codice front-end, ottenendo file statici serviti direttamente dal Backend Node.js.

In particolare nella cartella `frontend` è stato eseguito il seguente comando:

```
npm run build
```

che ha generato in output la cartella `dist` che contiene l'intera applicazione frontend pronta per essere servita, tale cartella va spostata nella cartella `backend`.

- Modificato il file `server.js`, il cuore del nostro back-end in modo tale da servire i file statici

```
app.use(express.static(path.join(__dirname, 'dist')));  
app.get('^(.*)$', (req,res) => {  
    res.sendFile(path.join(__dirname, 'dist', 'index.html'));  
});
```

Fatto ciò il progetto è pronto per il deploy effettivo, abbiamo scelto [render.com](#) come piattaforma per effettuarlo. In esso abbiamo:

1. Configurato il progetto come **Web Service**
2. Collegato la nostra repository github
3. Scelto la cartella di backend come directory radice
4. Configurato i comandi di build e start: `npm install` e `npm start`
5. Configurato tutte le **environment variables** necessarie al funzionamento della nostra app

KEY	VALUE
CLIENT_URL	.....
PORT	.....
REDIS_HOST	.....
REDIS_PASSWORD	.....
REDIS_PORT	.....
SECRET_JWT	.....

Fatto tutto ciò l'applicazione è stata deployata effettivamente, però c'è un ultimo particolare da gestire:

- [render.com](#) utilizza il protocollo HTTPS e siccome Janus è stato configurato con HTTP base potrebbe sorgere il problema del Mixed Content, ovvero dove due end-point vogliono comunicare tra loro ma con protocolli non compatibili

La soluzione più immediata sarebbe stata comprare un dominio con annessi certificati da utilizzare all'interno della macchina virtuale per poter abilitare HTTPS per Janus.

Abbiamo dovuto però implementare un **escamotage** per poter far parlare correttamente il Media server Janus e la nostra web app deployata senza dover acquistare nessun dominio, ovvero abbiamo cambiato l'architettura all'interno della nostra VM Oracle, che ora non contiene più solo il Media Server ma anche un Proxy.

## Soluzione Architetturale: SSL Termination Proxy

Abbiamo fatto uso di 3 componenti:

1. **Magic DNS** ([sslip.io](#)): Utilizzo di un servizio DNS dinamico che risolve un hostname basato sull'IP (es. [indirizzo\\_ip\\_pubblico\\_macchina.sslip.io](#)) verso l'IP stesso. Questo fornisce un FQDN (Fully Qualified Domain Name) valido.
2. **Let's Encrypt**: Generazione di un certificato SSL valido e riconosciuto globalmente per il dominio [sslip.io](#).
3. **Nginx come Reverse Proxy**: Nginx gestisce la terminazione SSL (ascolta su 443 HTTPS) e instrada il traffico internamente verso Janus (che rimane in HTTP su localhost).

All'interno della VM abbiamo:

1. Installato Nginx e il client Certbot per la gestione automatizzata dei certificati.

```
sudo apt install nginx certbot python3-certbot-nginx
```

2. Utilizzato `sslip.io` per richiedere un certificato per l'IP pubblico del server.

```
sudo certbot --nginx -d indirizzo_ip_pubblico_macchina.sslip.io
```

N.B Certbot configura automaticamente i percorsi dei certificati nel file di config di Nginx

3. Configurato il proxy Nginx, tamite il file `default` presente nella directory "/etc/nginx/sites-available" andando a specificare:

- a. il nome del server: "server\_name indirizzo\_ip\_macchina.sslip.io"

- b. la location, ovvero l'endpoint per l'API RESTful di Janus

```
location /janus {  
    proxy_pass http://127.0.0.1:8088/janus;  
    proxy_http_version 1.1;  
    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;  
}
```

4. Riavviato Nginx tramite il comando `sudo systemctl restart nginx`

Invece per quanto riguarda il codice della nostra web app abbiamo dovuto solo cambiare l'indirizzo IP precedente per connetterci al Media Server, con quello nuovo appena ottenuto.

Fatto ciò l'applicazione è deployata e funzionante.