RUGGINE

Sistema di chat real-time WebSocket in Rust e React

Introduzione

Ruggine è un app di chat in tempo reale pensata per conversazioni testuali semplici e veloci, con inviti e gruppi. Offre un backend ad alte prestazioni e un'interfaccia intuitiva.

Funzionalità principali:

- Login con username univoco (blocco dei duplicati)
- Visualizzazione utenti attivi
- Inviti a chat private e creazione di gruppi
- Accettazione/rifiuto inviti ed ingresso in chat
- Messaggi in tempo reale tra i membri della chat
- Notifiche sullo stato della chat
- Cambio stato (disponibile/in chat) e uscita dalla chat

Architettura

Webapp con architettura client - server:

- Client (frontend): React con Vite
- Server (backend): Rust con axum framework

Librerie e tecnologie utilizzate:

- Tokio: Runtime asincrono
- WebSocket: Comunicazione real-time

Modelli - overview (1)

Struct principali utilizzate:

- **User**
- **ChatMessage**: messaggio inviato nella chat tra utenti
- **WebSocketMessage**: wrapper comunicazione client-server
- **ChatInvite**: invito per entrare in chat
- **ChatType (enum):** definisce il tipo di chat (privata o di gruppo)
- **MessageType (enum):** definisce tutte le tipologie di messaggi possibili (Login, ChatMessage, ChatInvite, UserJoin etc.)

Le restanti struct si trovano all'interno del file: *type.rs*

Modelli (2)

```rust
pub struct User {
    pub username: String,
    pub is_available: bool,
    pub chat_id: Option<String>,
}
```

User

```rust
pub struct ChatMessage {
    pub id: Uuid,
    pub chat_id: Option<String>,
    pub username: String,
    pub content: String,
    pub timestamp: chrono::DateTime<chrono::Utc>,
    pub chat_type: ChatType,
}
```

ChatMessage

```rust
pub enum ChatType {
    Private { target: String },
    Group { members: Vec<String> },
    System,
}
```

ChatType

Modelli (3)

Enum molto importante utilizzato per discriminare le diverse tipologie di messaggio:

```rust
pub enum MessageType {
    Login,
    LoginSuccess,
    LoginError,
    ChatMessage,
    UserJoined,
    UserLeft,
    UserStatusChanged,
    UsersList,
    ChatInvite,
    ChatInviteResponse,
    ChatReady,
    AloneInChat,
    ChatUsersCount,  //aggiornamenti conteggio utenti chat
    ChatAbandoned,   // notifica abbandono definitivo chat privata
    ChatInvalidated, // invalida ChatReady obsolete
    Error,
}
```

Stato globale condiviso

Campi AppState:

- **connected_users:** HashMap di ConnectedUser
  - **ConnectedUser:** contine la struct User, canale websocket dell'utente (sender) **e** l'id della sessione
- **total_cpu_time**: Duration
  - monitoring tempo totale utilizzo CPU
- **chat_tracking**: HashMap di ChatUsersCount
- **private_chats_with_both_users**: HashSet<String>

Caratteristiche:

- Ogni elemento incapsulato da Arc<Mutex> per la condivisione sicura
- Tratto Clone per la distribuzione

WebSocket (1) - protocollo e motivazioni

Cos'è WebSocket:

- Protocollo di comunicazione che consente connessioni bidirezionali e persistenti tra client e server in tempo reale

Perché (vantaggi rispetto ad HTTP):

- **Real-time**: Messaggi istantanei senza polling
- **Stateful:** Mantiene lo stato della connessione
- **Persistent:** Connessione sempre aperta
- **Minimal overhead:** Headers solo nell'handshake iniziale

WebSocket (2) - implementazione nel progetto

Entry Point:

```rust
pub async fn websocket_handler(ws: WebSocketUpgrade, State(state): State<AppState>) -> Response {
    ws.on_upgrade(move |socket| handle_socket(socket, state))
}
```

- **Http upgrade**: da richiesta REST a connessione WebSocket
- **State Injection**: AppState condiviso

Protocollo standard:

```json
{
    "message_type": "ChatMessage",
    "data": "{\"content\":\"Ciao!\",\"username\":\"alice\"}"
}
```

message_type: Enum per routing

data: Payload JSON serializzato

WebSocket (3) - implementazione nel progetto

Loop principale:

```
loop {
    tokio::select! {
        maybe_out = rx.recv() => { /* Invio al client */ }
        incoming = receiver.next() => { /* Ricezione dal client */ }
    }
}
```

- Invio dei messaggi verso il client
- Gestione dei messaggi ricevuti dal client (in base al MessageType)

WebSocket (4) - gestione messaggi frontend

```
const connectWebSocket = () => {

    //si attiva ogni volta che il server invia un messaggio al client
    ws.onmessage = (event) => {
      try {
        const wsMessage = JSON.parse(event.data);

        switch (wsMessage.message_type) {
```

Funzione nel frontend che gestisce l'arrivo dei messaggi dal backend e li processa in base al *message_type*

REST API - protocollo HTTP

Oltre al WebSocket per real-time, sono stati implementati endpoint HTTP REST per operazioni stateless.

Lista route:

- **GET /api/users**: Ritorna array JSON di tutti gli utenti attualmente connessi
- **POST /api/login**: Controlla se username è disponibile prima di aprire WebSocket
- **POST /api/users/:username/availability**: Aggiorna stato disponibilità utente (available/busy)

Business logic - panoramica

- Tre moduli principali:
  - user.rs: login/stato utente/sessione
  - invites.rs: inviti (privati e di gruppo) e risposte
  - chat.rs: invio/ricezione messaggi e fan-out ai membri

- Scambio su WebSocket tramite MessageType e payload definiti in types.rs
- Stato condiviso in AppState (state.rs): utenti connessi, membership chat, contatori

user.rs: gestione utente e sessioni

Scopo: Gestisce il ciclo di vita utente (login, aggiornamento stato, cleanup).

Funzionalità:

- **broadcast_user_joined**: annuncia a tutte le connessioni che un nuovo utente è entrato
- **broadcast_user_status_changed**: notifica a tutti il cambio di stato di un utente (available/inChat, chatId)
- **send_users_list_to_all**: invia a tutti la lista completa degli utenti connessi
- **send_users_list**: invia la lista utenti a un singolo destinatario (ad es. nuovo connesso)
- **broadcast_to_all**: spedisce un identico messaggio a tutte le connessioni attive

chat.rs: gestione invio messaggi (1)

Scopo: Invio messaggi in chat solo ai corretti destinatari

Funzionalità:

- **broadcast_chat_message**: invio di un messaggio a tutti gli utenti nella stessa chat del sender
- **broadcast_user_left**: Gestione abbandono chat con notifiche sistema

Il corretto indirizzamento dei messaggi avviene attraverso il campo *chat_id*, esso identifica univocamente una chat e ogni utente *busy* ne ha uno associato.

In alternativa, in caso di errori, anche *chat_type* contiene (oltre all'indicazione sul tipo di chat) i destinatari del messaggio e quindi può essere utilizzato per l'inoltro di essi.

chat.rs: passaggi invio messaggio (2)

broadcast_chat_message (detail):

```rust
// 1. Serializzazione messaggio
let message = WebSocketMessage {
    message_type: MessageType::ChatMessage,
    data: serde_json::to_string(chat_msg).unwrap(),
};

// 2. Chat ID Resolution (con fallback)
let target_chat_id = if let Some(explicit_chat_id) = chat_msg.chat_id.clone() {
    Some(explicit_chat_id)  // Usa chat_id esplicito
} else {
    // Fallback: prendi chat_id del sender
    users.get(sender_username).and_then(|u| u.user.chat_id.clone())
};

// 3. Filtering intelligente
for (_, connected_user) in users.iter() {
    if let Some(user_chat_id) = &connected_user.user.chat_id {
        if user_chat_id == &chat_id {
            let _ = connected_user.sender.send(message_json.clone());
        }
    }
}
```

1. Serializzazione messaggio nella Struct WebSocketMessage
2. Ottenimento *chat_id* dal messaggio
   a. gestione casista in cui il *chat_id* non sia presente
3. Invio del messaggio a tutti gli utenti con stesso *chat_id*

Invio messaggi nel frontend

```
const chatMessage = {
  id: crypto.randomUUID(),
  chat_id: chatId,
  username: user.username,
  content: content.trim(),
  timestamp: new Date().toISOString(),
  chat_type: chat_type_obj
};

const wsMessage = {
  message_type: 'ChatMessage',
  data: JSON.stringify(chatMessage)
};

try {
  wsRef.current.send(JSON.stringify(wsMessage));
```

La funzione *sendMessage*, prende il contenuto della textbox nella chat e ci crea un messaggio con tutte le informazioni necessarie.

Il messaggio viene poi serializzato e inviato al backend.

invites.rs: gestione inviti e risposte

Scopo: Gestisce l'intero ciclo "invito → risposta" per chat private e di gruppo.

Funzionalità:

- send_chat_invite : recapita l'invito ai destinatari corretti e inizializza il tracking della chat
- handle_invite_response : applica l'esito della risposta all'invito (accettato/rifiutato), notifica le parti interessate e aggiorna tracking/chat

Creazione invito nel frontend

```
const chatId = crypto.randomUUID();
const invite = {
  id: crypto.randomUUID(),
  chat_id: chatId,
  from: user.username,
  from_session_id: sessionIdRef.current,
  chat_type: chatType === 'private'
    ? { Private: { target: targetUser } }
    : { Group: { members: members } },
  message: message || `${user.username} ti ha invitato in una chat
  timestamp: new Date().toISOString()
};
```

Una volta selezionato il tipo di chat e i membri viene generato un *chat_id* per quella chat e in seguito l'invito ad essa. L'invito viene poi inviato al backend che lo inoltrerà correttamente

Ciclo di vita stato utente

Quando un utente effettua il login viene aggiunto allo stato globale, in particolare nei *connected_users.* Inizialmente il campo *is_available* sarà *true* e *chat_id* sarà *null*.

Quando un utente entra in chat:

- is_available = false
- chat_id = id della chat a cui partecipa

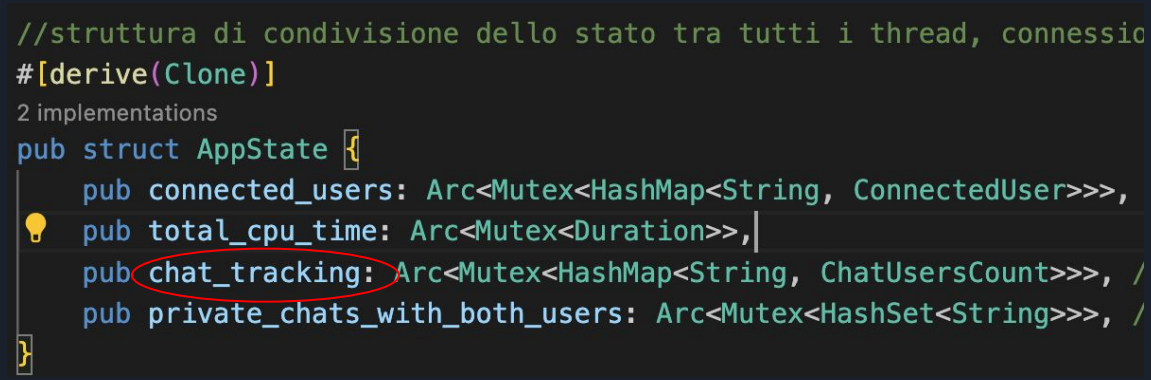
Quando un utente abbandona la chat le 2 variabili ritornano allo stato iniziale (*false*, *null*)

Quando un utente effettua il logout esso viene rimosso dallo stato globale

Ogni qual volta avviene un cambio di stato da parte di un utente, ciò viene notificato a tutti con un messaggio broadcast di tipo *UserStatusChanged*.

Tracking

L’applicazione tiene traccia di alcune informazioni essenziali per ciascuna chat come un identificatore univoco e una lista di tutti gli utenti entrati in chat e invitati



Tracking (2)

tracking.rs

Per manipolare chat_tracking si usano specifiche funzioni presenti nel file tracking.rs come `init_chat_tracking` per inizializzare la struttura o `add_user_to_chat_tracking` per inserire un utente che si è collegato ad una chat.

Performance

L'applicazione tiene traccia del tempo complessivo di cpu utilizzato dall'avvio nello stato.

Protetto da un Mutex per garantire la sicurezza di accesso concorrente.

```rust
//struttura di condivisione dello stato tra tutti i thread, connessio
#[derive(Clone)]
2 implementations
pub struct AppState {
    pub connected_users: Arc<Mutex<HashMap<String, ConnectedUser>>>,
    pub total_cpu_time: Arc<Mutex<Duration>>,
    pub chat_tracking: Arc<Mutex<HashMap<String, ChatUsersCount>>>, /
    pub private_chats_with_both_users: Arc<Mutex<HashSet<String>>>, /
}
```

Performance (2)

la funzione `update_cpu_time` aggiorna il valore in base a un istante iniziale richiesto come parametro

```rust
//misura e accumula tempo di cpu
pub fn update_cpu_time(total_cpu_time: Arc<Mutex<Duration>>, start: Instant) {
    let elapsed: Duration = start.elapsed();
    let mut total: MutexGuard<'_, Duration> = total_cpu_time.lock().unwrap();
    *total += elapsed;
}
```

Performance (3)

Consentendo di misurare il tempo di utilizzo di CPU negli intervalli di codice sincroni

```rust
let mut start: Instant = Instant::now();
let message: WebSocketMessage = WebSocketMessage {
    message_type: MessageType::ChatInvite,
    data: serde_json::to_string(invite).unwrap(),
};
let message_json: String = serde_json::to_string(&message).unwrap();
//aggiorna il tempo di CPU//
update_cpu_time(state.total_cpu_time.clone(), start);
let users: MutexGuard<'_, HashMap<String, …>> = state.connected_users.lock().unwrap();
```

Performance (4)

Il sistema, tramite la funzione `start_log` genera all'avvio un thread che ogni 2 minuti scrive su un file di testo copiando il valore del tempo di utilizzo dallo stato e rimanendo in stato di sleep il resto del tempo (in modo da non consumare risorse)

```rust
pub fn start_log(duration: Arc<Mutex<Duration>>) {
    tokio::spawn(future: async move {
        loop {
            // Copia il valore sotto lock e rilascia subito il mutex
            let secs: f64 = {
                let d: MutexGuard<'_, Duration> = duration.lock().unwrap();
                d.as_secs_f64()
            };
```

```rust
// Apri il file in append e scrivi
match OpenOptions::new() OpenOptions
    .write(true) &mut OpenOptions
    .create(true) &mut OpenOptions
    .append(true) &mut OpenOptions
    .open(path: "Log/cpu_log.txt") impl Future<Output = Result<…, …>>
    .await
{
    Ok(mut file: File) => {
        if let Err(e: Error) = file.write_all(src: line.as_bytes()).await {
            eprintln!("Errore scrittura file: {}", e);
        }
    }
```

```rust
tokio::time::sleep(Duration::from_secs(120)).await;
```

Test

Librerie utilizzate : tokio, axum, tokio-tungstenite, serde/serde_json, uuid, chrono

Test implementati

- Test 1 — Login duplicato rifiutato
  - Setup: 2 client → entrambi inviano LoginRequest con lo stesso username.
  - Atteso: il primo riceve LoginSuccess; il secondo riceve LoginError e non viene registrato.
- Test 2 — Invito privato consegnato
  - Setup: A e B si collegano; A invia ChatInvite verso B.
  - Atteso: B riceve un messaggio MessageType::ChatInvite con il payload dell'invito.

Test (2)

- **Test 3 — Messaggio di gruppo a tutti i membri**
  - Setup: A, B, C effettuano login; tutti segnalano UserStatusChanged con lo stesso chatId; A invia un ChatMessage al gruppo.
  - Atteso: B e C ricevono il messaggio

- **Test 4 — Messaggio consegnato solo ai membri della stessa chat**
  - Setup: quattro client effettuano login. alice e bob impostano UserStatusChanged con chatId="group-1"; carol e dave impostano UserStatusChanged con chatId="group-2". Si attende conferma per ciascun utente.
  - Atteso: alice invia un ChatMessage per chatId="group-1". bob (membro di group-1) riceve il messaggio; carol e dave (membri di group-2) non ricevono alcun ChatMessage entro il timeout.