Panoramica dei Crate del Progetto Ruggine

Il progetto Ruggine si basa su un ecosistema completo di 23 crate specializzati, ognuno selezionato per fornire funzionalità specifiche nell'architettura di una chat sicura e performante. Questa panoramica tecnica analizza l'integrazione strategica di ogni componente nell'infrastruttura del sistema.

Architettura dei Componenti Principali

Framework & Runtime

tokio v1.37 - Runtime asincrono completo con features "full"

iced v0.12 - Framework GUI cross-platform con integrazione tokio

Security & Crypto

argon2 v0.5 - Hashing sicuro password con protezione anti-brute force

ring v0.17 - Crittografia AES-256-GCM per cifratura messaggi

rustls v0.21 - Stack TLS/SSL completo per connessioni sicure

tokio-rustls v0.24 - Integrazione rustls con tokio

rustls-pemfile v1.0 - Gestione certificati PEM

base64 v0.22 - Encoding/decoding base64

md5 v0.7 - Hashing MD5

rand v0.8 - Generazione numeri casuali crittografici

keyring v1.1 - Gestione credenziali sistema

Network & Communication

tokio-tungstenite v0.21 - WebSocket asincrono per comunicazione real-time

futures-util v0.3 - Stream management e channel splitting

url v2.5 - Parsing e manipolazione URL

Data & Persistence

sqlx v0.7 - Database toolkit con type safety e connection pooling SQLite

redis v0.24 - Caching distribuito e pub/sub per scalabilità multi-server

serde v1.0 - Serializzazione/deserializzazione con derive macros

serde_json v1.0 - Supporto JSON per serde

chrono v0.4 - Gestione date e timestamp con serde

Utilities & System

uuid v1.0 - Generazione UUID v4 con supporto serde

anyhow v1.0 - Error handling semplificato

log v0.4 - Logging framework

env_logger v0.10 - Logger per variabili ambiente

dotenvy v0.15 - Caricamento file .env

sysinfo v0.30 - Informazioni sistema e monitoraggio

Architettura Database per Sistema Chat Crittografato

Il sistema implementa un database relazionale sofisticato composto da **14 tabelle interconnesse**, progettato specificamente per supportare un'applicazione di messaggistica con crittografia end-to-end. L'architettura garantisce sicurezza, scalabilità e performance ottimali per la gestione di comunicazioni private.

Core Autenticazione

- **auth** - Sistema autenticazione
- **sessions** - Gestione sessioni
- **session_events** - Log eventi
- **users** - Profili utente

Messaggistica Sicura

- **encrypted_messages** - Messaggi crittografati
- **deleted_chats** - Cronologia eliminazioni
- **user_encryption_keys** - Chiavi personali

Gestione Gruppi

- **groups** - Metadati gruppi
- **group_members** - Appartenenze
- **group_invites** - Sistema inviti
- **group_encryption_keys** - Crittografia gruppo

Social Network

- **friendships** - Relazioni amicizia
- **friend_requests** - Richieste pendenti
- **sqlite_sequence** - Gestione ID

Implementazione Database con Rust e SQLx

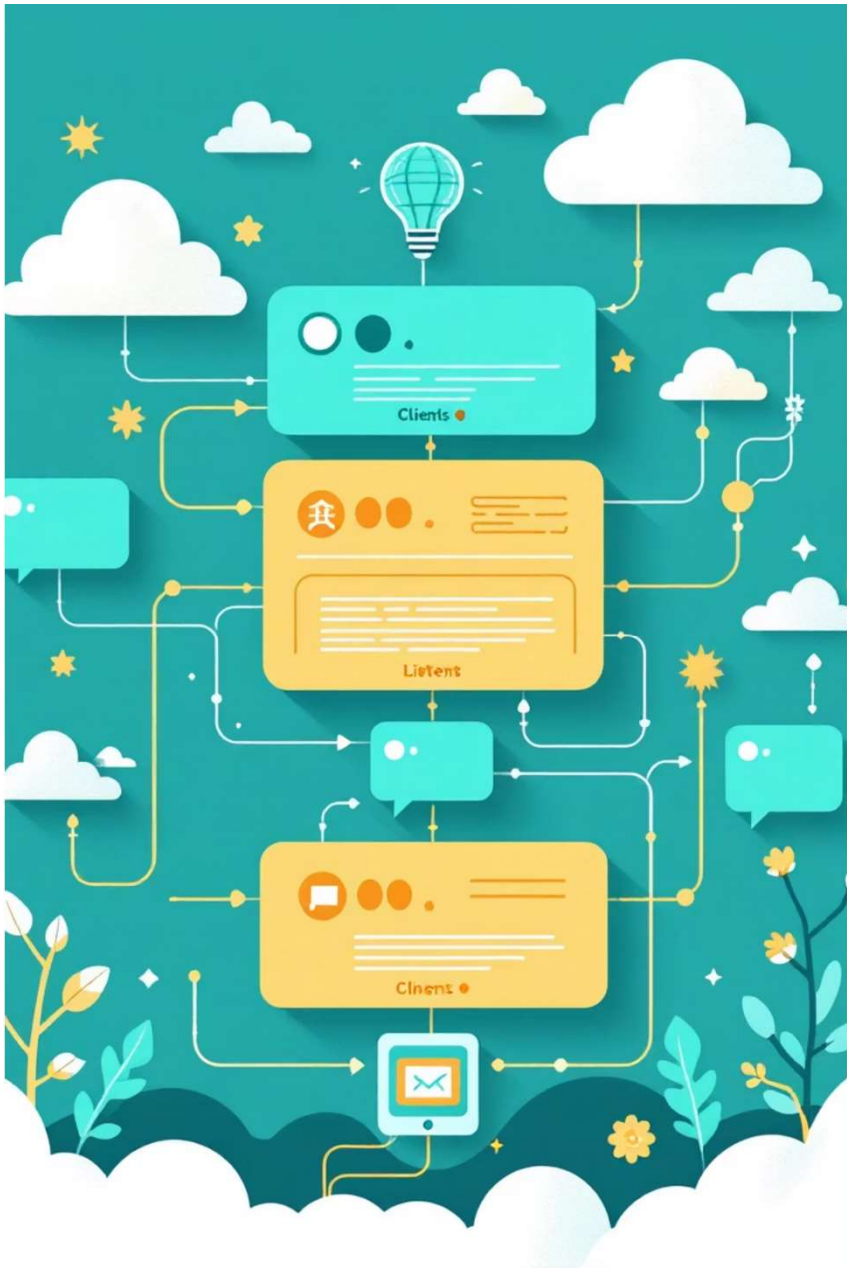
Approccio Migrazionale Programmatico

Il database viene inizializzato e mantenuto attraverso un sistema di **migrazioni programmatiche** implementato in Rust utilizzando la libreria SQLx. Questo approccio garantisce controllo versionale dello schema e deployment automatizzato.

```
pub async fn migrate(&self) -> Result<(), sqlx::Error> {
// Esecuzione migrazioni sequenziali
// Gestione errori type-safe
// Rollback automatico in caso di fallimento
}
```

Architettura di Comunicazione Client-Server

Un'analisi approfondita dell'implementazione di un sistema di comunicazione asincrono basato su pattern avanzati di Rust per applicazioni real-time distribuite.



Componenti del Sistema

GUI (app.rs)

Interfaccia utente costruita con Iced, responsabile della gestione degli eventi e della presentazione dei dati. Comunica con il ChatService attraverso chiamate asincrone.

ChatService

Servizio persistente che gestisce la comunicazione client. Mantiene canali MPSC per l'invio di comandi e coordina le operazioni di rete asincrone.

Background Task

Task asincrono dedicato alla gestione continua della connessione TCP. Opera in un loop infinito per processare comandi e gestire le risposte dal server.

Server TCP

Server remoto che riceve e processa le richieste attraverso protocollo TCP. Fornisce risposte strutturate ai client connessi.

Struttura del ChatService

Definizione della Struct

```
pub struct ChatService {
    pub tx: Option<mpsc::UnboundedSender<
    (CommandType, oneshot::Sender<String>)>>,
    pub _bg: Option<tokio::task::JoinHandle<()>>,
}
```

Il servizio mantiene due componenti critici: un sender MPSC per la comunicazione con il background task e un handle per la gestione del task asincrono.

L'architettura sfrutta i pattern di concorrenza di Rust per garantire comunicazioni thread-safe e performance elevate attraverso canali tipizzati.

Pattern MPSC + Oneshot

1

MPSC Channel Setup

```
let (tx, mut rx) = mpsc::unbounded_channel::<(CommandType, oneshot::Sender<String>)>();
```

Canale Multi-Producer Single-Consumer che permette alla GUI di inviare multipli comandi mentre un solo background task li riceve sequenzialmente.

2

Oneshot per Ogni Comando

```
let (resp_tx, resp_rx) = oneshot::channel();tx.send((CommandType::SingleLine(cmd), resp_tx))?;
```

Ogni comando riceve un canale oneshot privato per la sua risposta specifica, garantendo che ogni richiesta sia associata alla corretta risposta.

3

Attesa Asincrona

```
let resp = resp_rx.await    .map_err(|_| anyhow!("channel closed"))?;
```

La GUI attende asincronamente la risposta senza bloccare il thread principale, mantenendo la reattività dell'interfaccia utente.

Background Task: Il Cuore del Sistema

1) Ricezione Comando

```
let (cmd_type, resp_tx) = match rx.recv().await {
Some(pair) => pair,
None => break,
};
```

Il task attende continuamente nuovi comandi dal canale MPSC, gestendo gracefully la chiusura del canale.

2) Invio TCP al Server

```
writer.write_all(cmd.as_bytes()).await?;
writer.write_all(b"\n").await?;
writer.flush().await?;
```

Il comando viene serializzato e inviato al server attraverso una connessione TCP bufferizzata per ottimizzare le performance di rete.

3) Lettura Risposta

```
let mut server_line = String::new();
reader.read_line(&mut server_line).await?;
```

Lettura asincrona della risposta del server utilizzando un buffer reader per efficienza nella gestione dei dati di rete.

4) Invio alla GUI

```
let _ = resp_tx.send(    server_line.trim().to_string());
```

La risposta viene inviata direttamente alla GUI attraverso il canale oneshot specifico per quel comando.

Flusso Completo di Comunicazione

1 GUI Inizializza Richiesta

La GUI invoca `send_command()` per inviare un comando specifico al server. Viene creato un canale oneshot per gestire la risposta asincrona di questa richiesta.

2 Invio tramite MPSC

Il comando viene inviato al Background Task utilizzando `self.tx.send()` attraverso il canale MPSC, permettendo comunicazione thread-safe tra componenti.

3 Ricezione Background Task

Il Background Task riceve il messaggio tramite `rx.recv()` nel suo loop asincrono continuo, mantenendo la connessione TCP sempre attiva.

4 Comunicazione TCP Server

Viene stabilita la comunicazione bidirezionale con il server: TCP write per l'invio del comando e TCP read per la ricezione della risposta elaborata.

5 Ritorno alla GUI

Il Background Task invia la risposta tramite `resp_tx.send()`. Il ChatService attende con `resp_rx.await` e restituisce il risultato alla GUI.

Vantaggi dell'Architettura

Thread Safety

I canali Rust garantiscono comunicazione sicura tra thread senza data races o condizioni critiche, eliminando errori di concorrenza.

Performance Asincrone

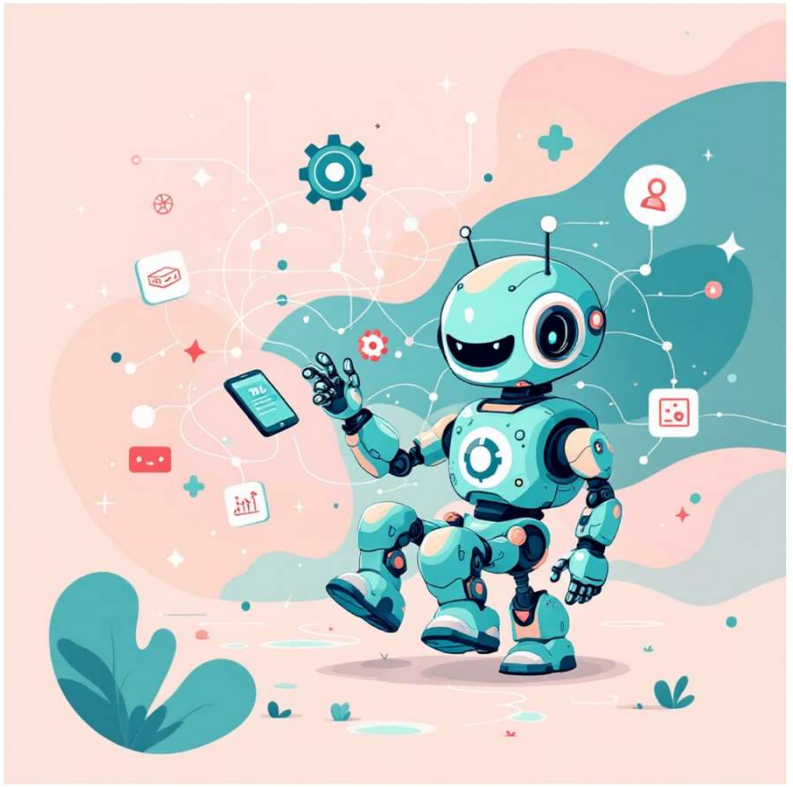
L'utilizzo di tokio e async/await permette di gestire migliaia di operazioni concorrenti senza bloccare thread del sistema operativo.

Connessione Persistente

Il background task mantiene una connessione TCP sempre attiva, riducendo latenza e overhead di stabilimento connessioni ripetute.

Reattività GUI

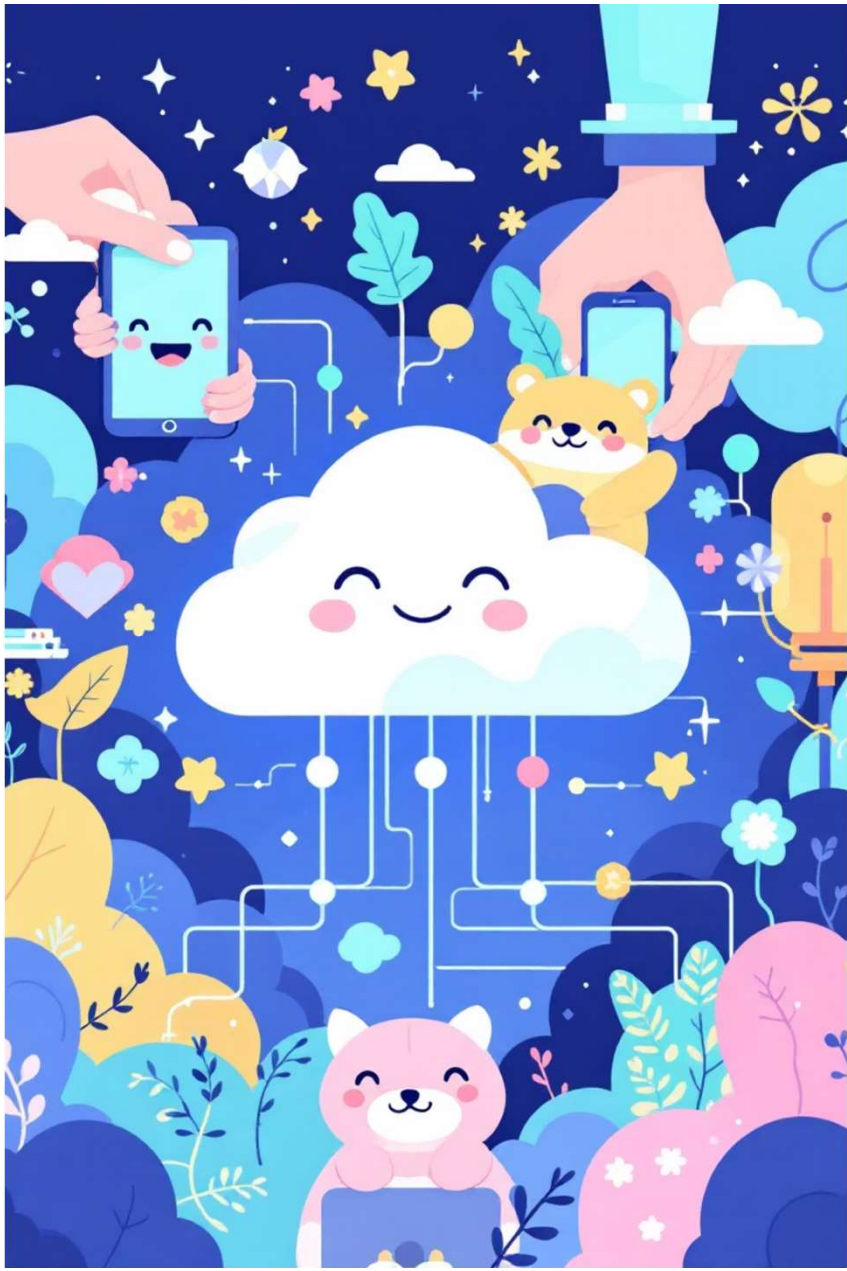
La separazione tra interfaccia utente e comunicazione di rete garantisce che l'UI rimanga sempre responsiva durante le operazioni di rete.



Risultato: Un'architettura robusta e scalabile che combina la sicurezza di Rust con pattern asincroni avanzati per comunicazioni client-server efficienti.

Gestione Sessioni e Presenza: Architettura per Single-Session Guarantee

Un sistema robusto per garantire una sola sessione attiva per utente, implementando meccanismi di autenticazione, validazione e gestione della presenza in tempo reale per applicazioni backend critiche.



Login con Garanzia Single-Session

Flusso Client

Il client invia una richiesta `/login` con credenziali utente al server per inaiziare una nuova sessione autenticata.

Processo Server

1) Pulizia Sessioni

Elimina tutte le righe esistenti nella tabella `sessions` per il `user_id` specifico

2) Nuova Sessione

Inserisce una nuova riga con token generato e imposta `users.is_online = 1`

3) Event Logging

Registra evento `login_success` e esegue commit di tutte le operazioni

4) Gestione Presenza

Chiama `presence.kick_all(user_id)` per disconnettere vecchie connessioni e registra nuova presenza

Auto-login e Logout: Gestione del Ciclo di Vita

Auto-login al Startup

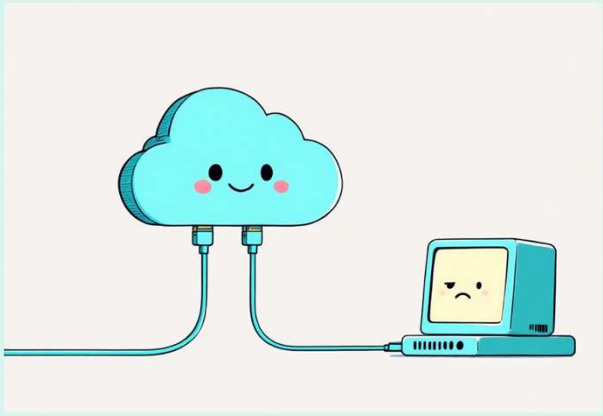
Il client utilizza il token salvato per inviare `/validate_session` tramite `ChatService` persistente. Se la validazione è positiva, il server registra la presenza **senza eseguire kick**, mantenendo la sessione corrente attiva e impostando `is_online = 1`.

Logout Esplicito

Durante il logout, il client invia `/logout` e il server verifica il token, chiama `auth::logout` per eliminare le sessioni, imposta `users.is_online = 0` e registra l'evento. Tutte le connessioni attive vengono forzatamente chiuse tramite `presence.kick_all`.

Quit Inaspettato

Quando la connessione TCP si chiude inaspettatamente (`read_line -> 0`), il server esegue cleanup della presenza. Se non ci sono più connessioni attive, imposta `is_online = 0` e registra evento `quit`, **mantenendo la sessione valida** per futuri auto-login.



Implementazione Tecnica: Validazione e Kick Management

Funzione validate_session

```
pub async fn validate_session(db: Arc<Database>, session_token: &str) ->
Option<String> {
     let now = chrono::Utc::now().timestamp();
     let row = sqlx::query(
          "SELECT user_id FROM sessions
          WHERE session_token = ? AND expires_at > ?")
     .bind(session_token)
     .bind(now)
     .fetch_optional(&db.pool)
     .await.ok()?;
     row.map(|r| r.get::<String,_>("user_id"))
}
```

La funzione verifica la validità del token confrontando timestamp corrente con `expires_at` e restituisce `user_id` se valido.

Sistema kick_all

```
pub async fn kick_all(&self, user_id: &str) -> usize {

     let mut map = self.inner.lock().await;
     if let Some(vec) = map.remove(user_id) {
          let count = vec.len();
          for tx in vec {
               let _ = tx.send(());
          }
          count
     } else {
     0
     }
}
```

Gestisce HashMap delle connessioni attive, inviando segnali di disconnessione a tutti i canali registrati per l'utente specifico.

Implementazione WebSocket

Punti Chiave:

- **Tokio:** Runtime asincrono per gestire la concorrenza.
- **Tungstenite:** Implementazione del protocollo WebSocket.
- **Canali MPSC:** Fondamentali per la comunicazione tra task e thread.
- **Autenticazione:** Sicurezza e gestione delle sessioni utente.

Questa base solida consente di costruire funzionalità avanzate come chat di gruppo e scalabilità orizzontale con l'integrazione di sistemi di caching distribuiti come Redis.

Architettura WebSocket Asincrona

La nostra applicazione di chat si basa su WebSocket per una comunicazione bidirezionale real-time, gestita in modo asincrono per massimizzare l'efficienza. Utilizziamo **Tokio** per il runtime asincrono e **Tungstenite** per l'implementazione del protocollo WebSocket.

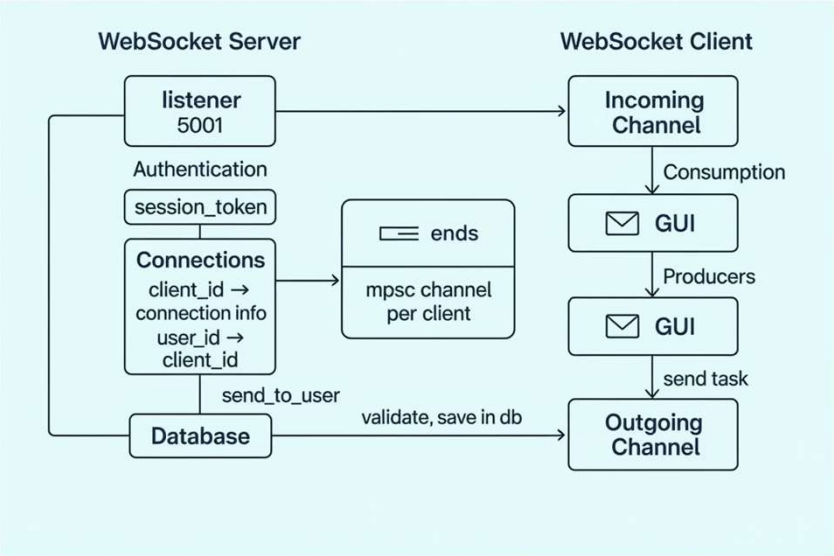
Tokio-Tungstenite v0.21

Dove: `src/server/websocket.rs`, `src/client/services/websocket_client.rs` **Per cosa:** WebSocket real-time per chat, async message streaming.

Il server gestisce in parallelo l'invio e la ricezione dei messaggi grazie alla macro `tokio::select!`, che consente di ascoltare contemporaneamente più canali asincroni, garantendo una reattività immediata.

```
tokio::select! { Some(outgoing) = outgoing_rx.recv() => {
ws_sender.send(outgoing).await?;
} Some(incoming) = ws_receiver.next() => {
process_incoming(incoming).await;
}
}
```

Avvio del Server WebSocket e Gestione delle Connessioni



Il server WebSocket rimane in ascolto su una porta specificata (es. **5001**).

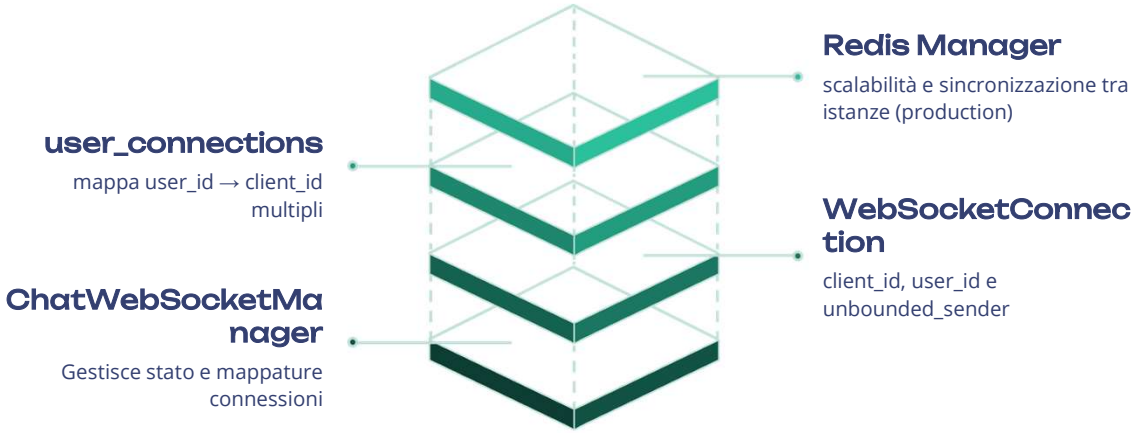
Ogni **nuova connessione** client viene accettata e gestita da un **task** `tokio::spawn` dedicato, garantendo che ogni client abbia la sua elaborazione **asincrona** e **indipendente**.

Dopo **l'handshake** WebSocket, la connessione passa attraverso un processo di **autenticazione** critico, che valida il client prima di stabilire una sessione di chat completa.

```
async fn start_websocket_server(...) -> anyhow::Result<()> {
    let listener = TcpListener::bind(addr).await?;
    while let Ok((stream, addr)) = listener.accept().await {
        tokio::spawn(async move {
        match tokio_tungstenite::accept_async(stream).await {
            Ok(ws_stream) => { /* Gestione connessione autenticata */ }
            Err(e) => { error!("Error during WebSocket handshake: {}", e);
            }
            }
        });
    }
Ok(())}
```

Gestione Connessioni Client lato Server

Il `ChatWebSocketManager` è il cuore della gestione delle connessioni sul server. Mantiene lo stato di tutti i client connessi e facilita la comunicazione tra di essi. Utilizza diverse mappe e canali per organizzare in modo efficiente i client.




```
// src/server/websocket.rs - Struttura per gestire i clientpub struct ChatWebSocketManager {
connections: Arc<Mutex<HashMap<ClientId, WebSocketConnection>>>, //MAPPA CLIENT: client_id → connection info
user_connections: Arc<Mutex<HashMap<UserId, ClientId>>>, //MAPPA UTENTI: user_id → client_id (lookup veloce)
message_broadcaster: broadcast::Sender<WebSocketMessage>, // BROADCASTER: per messaggi a tutti i client
redis_manager: Arc<Mutex<ConnectionManager>>, //per scalabilità multi-server futura
```

Autenticazione e Canali di Comunicazione Client

Al momento della connessione, il client tenta di autenticarsi inviando un `AuthMessage` contenente il proprio session token. Solo dopo un'autenticazione riuscita vengono stabiliti i canali di comunicazione per i messaggi in uscita (outgoing).

Canale Incoming (mpsc::UnboundedSender)

Produttori multipli: server con task paralleli per errori, timeout, parsing error. Anche se non possiede il **tx** del canale fisicamente, la gui ascolta i messaggi sul **websocket_receiver** e li inoltra nel canale mpsc con **handle_incoming_messages()**.

Consumatore singolo: La GUI dell'applicazione (ChatService) che che prende **rx** con **take_receiver())**

Canale Outgoing (mpsc::UnboundedSender)

Produttori multipli: GUI con task paralleli prendono **tx** da **try_connect()** per messaggi privati, di gruppo o comandi.

Consumatore singolo: Task interno al client che invia messaggi al server tramite **websocket_sender**.

```
let (outgoing_tx, mut outgoing_rx) = mpsc::unbounded_channel();

// ...
tokio::spawn(async move { while let Some(outgoing_msg) = outgoing_rx.recv().await { // Invia al server via WebSocket
}});
```

Architettura Multi-Producer, Single-Consumer (MPSC)

L'architettura di messaggistica è un esempio classico di pattern MPSC, utilizzando canali `tokio::sync::mpsc::UnboundedSender` e `Receiver`. Questo modello è fondamentale per gestire il flusso di messaggi in un'applicazione real-time, dove più componenti possono inviare e ricevere dati in modo concorrente.

- **Lato Client:** Diversi thread della GUI producono messaggi (outgoing) che un singolo task consuma per inviarli al server. Al contrario, il server e altri task producono messaggi (incoming) che la GUI consuma.
- **Lato Server:** Ogni client ha un suo canale dedicato (produttore) per i messaggi in arrivo dal server, consumato dal client stesso. Il broadcaster di messaggi opera anch'esso come MPMC, permettendo a più componenti di inviare messaggi a tutti i client.

Questa flessibilità è cruciale per la reattività e la scalabilità del sistema di chat.

Flusso di Invio Messaggi tra Client

L'invio di un messaggio da un Client A a un Client B segue un percorso ben definito attraverso il server, garantendo che i messaggi siano salvati e consegnati in modo affidabile.

1

1. Client A → Server

Invio del messaggio JSON tramite WebSocket, includendo tipo di chat, destinatario e contenuto.

2

2. Server Riceve

Il task di ricezione del server parsa il messaggio, lo salva nel database e prepara l'invio al destinatario.

3

3. Invio a Client B

Il `ChatWebSocketManager` trova la connessione di Client B e utilizza il suo sender MPSC dedicato.

4

4. Client B Riceve

Il task di invio di Client B riceve il messaggio dal suo canale MPSC e lo inoltra via WebSocket alla GUI.

```
if let Some(connection) = connections.get(client_id) { let json_message = serde_json::to_string(&message)?; let _ =
connection.sender.send(Message::Text(json_message));}
```

Crittografia End-to-End per App di Messaggistica

Benvenuti a questa presentazione tecnica sulla crittografia End-to-End (E2E) per applicazioni di messaggistica. Esploreremo i principi fondamentali e il flusso dettagliato di come i messaggi vengono protetti, garantendo la massima privacy per i vostri utenti.

Architettura a 3 Livelli della Crittografia E2E

La nostra architettura di crittografia E2E si basa su un sistema robusto a tre livelli, progettato per bilanciare sicurezza, efficienza e scalabilità. Questo approccio garantisce che ogni messaggio sia protetto individualmente, pur mantenendo una gestione centralizzata e sicura delle chiavi.

Master Key (Server)

La chiave primaria custodita sul server, base di tutta la sicurezza.

Chat Key (Per Conversazione)

Chiave unica generata per ogni coppia di utenti/conversazione, derivata dalla Master Key.

Crittografia Messaggio (AES-256-GCM)

L'algoritmo di crittografia simmetrica per proteggere il contenuto di ogni singolo messaggio.

Questo sistema a livelli assicura che anche se un livello di sicurezza dovesse essere compromesso, i dati degli altri livelli rimarrebbero protetti, riducendo significativamente la superficie di attacco.

Flusso di Crittografia: Dalla Master Key al Messaggio

Fase 1: Setup Master Key

Allo startup del server, viene caricata una chiave Master Key, fondamentale per l'intero sistema. Questa chiave non viene mai trasmessa ai client e serve come radice di fiducia per tutte le derivazioni successive.

```
let master_key = CryptoManager::load_master_key_from_env();
```

Fase 2: Generazione Chat Key

Quando due utenti, ad esempio Alice e Bob, iniziano a chattare, viene generata una chiave di chat unica per la loro conversazione. Questa chiave è derivata dalla Master Key e dai loro ID utente, garantendo che solo i partecipanti possano ricrearla.

```
let participants = vec!["alice".to_string(),
"bob".to_string()];let chat_key =
CryptoManager::generate_chat_key(&participants,
&master_key);// Internamente: SHA-256(master_key +
"alice" + "bob")
```

L'ordine dei partecipanti è cruciale per la generazione della chiave, assicurando che la stessa coppia di utenti generi sempre la stessa Chat Key.

Crittografia e Decrittografia del Messaggio

Fase 3: Crittografia (Lato Alice)

Quando Alice invia un messaggio, il sistema recupera i partecipanti, genera la Chat Key specifica e poi utilizza l'algoritmo AES-256-GCM per crittografare il testo. Un `nonce` (numero usato una sola volta) casuale viene generato per ogni crittografia, prevenendo attacchi di replay.

```
let chat_key = generate_chat_key(&participants, &master_key);let (ciphertext, nonce) = encrypt_message("Ciao Bob", &chat_key);let encrypted_data = serde_json::json!({    "ciphertext": base64::encode(&ciphertext),    "nonce": base64::encode(&nonce)});
```

Il messaggio crittografato e il nonce vengono poi memorizzati in un database, pronti per essere recuperati dal destinatario.

Fase 4: Decrittografia (Lato Bob)

Quando Bob riceve un messaggio, il processo è inverso. Bob ricostruisce la stessa Chat Key usando gli stessi partecipanti e la Master Key (tramite il server). Recupera il ciphertext e il nonce dal database e li utilizza per decrittografare il messaggio.

```
let chat_key = generate_chat_key(&participants, &master_key);let encrypted_data = get_from_database();let ciphertext = base64::decode(encrypted_data["ciphertext"]);let nonce = base64::decode(encrypted_data["nonce"]);let plaintext = decrypt_message(&ciphertext, &nonce, &chat_key);// Risultato: "Ciao Bob"
```

La simmetria del processo garantisce che solo chi possiede la Chat Key corretta (ovvero i partecipanti alla conversazione) possa leggere il contenuto in chiaro del messaggio.

Gestione Visualizzazione Messaggi in Rust

Esploriamo l'architettura completa per la gestione dei messaggi in un'applicazione chat sviluppata in Rust, analizzando la struttura dati AppState e il flusso end-to-end dall'input utente alla visualizzazione finale.

Architettura dello Stato e Flusso Messaggi

Struttura AppState

La struct AppState centralizza la gestione dei messaggi utilizzando HashMap per chat private (user_id → Vec<ChatMessage>) e di gruppo (group_id → Vec<ChatMessage>), con HashSet dedicati per tracciare gli stati di caricamento e un campo per l'input corrente.

Input Utente

Quando l'utente invia un messaggio, viene immediatamente aggiunto alla collezione locale private_chats[user] per garantire feedback istantaneo. Il campo di input si svuota e l'UI si aggiorna senza attendere la conferma del server, assicurando massima responsività.

Comunicazione WebSocket

La funzione send_private_message() gestisce l'invio in background al server, che salva il messaggio nel database e lo inoltra al destinatario tramite WebSocket. Il client destinatario intercetta WebSocketMessageReceived(IncomingChatMessage) e aggiorna il proprio stato locale.

Rendering GUI

build_messages_area() legge i messaggi da private_chats e utilizza create_message_bubble() per il rendering. I messaggi locali appaiono allineati a destra, quelli ricevuti a sinistra, con stili grafici distinti per mittente e destinatario.

Sistema di Performance Logging di Ruggine

Il sistema di performance logging di Ruggine rappresenta una soluzione avanzata per il monitoraggio continuo delle prestazioni del server chat. Questo modulo dedicato raccoglie metriche critiche di sistema e database in tempo reale, generando automaticamente il file `ruggine_performance.log` con aggiornamenti ogni 2 minuti.

Componenti Principali

Modulo Performance Logger integrato in `src/utils/performance.rs` con inizializzazione automatica dal server principale.

Output Strutturato

File CSV con metriche temporizzate: utenti attivi, gruppi, messaggi totali e utilizzo CPU medio del sistema.

Task Asincrono

Esecuzione non-bloccante tramite Tokio spawn, garantendo continuità operativa del server principale senza interferenze.

Implementazione Tecnica e Funzionalità Avanzate

L'implementazione del sistema utilizza pattern avanzati di programmazione Rust per garantire robustezza, efficienza e scalabilità. Il modulo opera come task indipendente con gestione intelligente degli errori e ottimizzazioni per ambienti di produzione.

01

Inizializzazione Sistema

Creazione automatica del file log, scrittura header CSV condizionale e inizializzazione del sistema di monitoraggio `sysinfo`.

02

Raccolta Metriche

Query database asincrone per utenti attivi, gruppi e messaggi con calcolo CPU usage medio su tutti i core disponibili.

03

Persistenza Dati

Scrittura atomica nel file CSV con timestamp localizzato UTC+2 e flush immediato per garantire integrità dei dati.

Gestione Errori Avanzata

Il sistema implementa una strategia di error handling resiliente che mantiene l'operatività anche in caso di errori database o I/O. Utilizza valori sentinella (-1) per errori di query e logging dettagliato per troubleshooting.

> Ogni 120 secondi, il sistema raccoglie automaticamente le metriche e le persiste nel formato CSV, garantendo continuità operativa senza impatto sulle performance del server principale.

Output Esempio

```
2025-09-16 19:30:43 UTC, 2, 5, 147, 23.4%2025-09-16 19:32:43 UTC, 3, 5, 152,
18.7%2025-09-16 19:34:43 UTC, 1, 5, 159, 15.2%
```

120s

Intervallo

Frequenza campionamento metriche

4

Metriche Core

Utenti, gruppi, messaggi, CPU

Dimensioni degli Eseguibili della Piattaforma Report

Un'analisi dettagliata delle dimensioni dei file eseguibili per diverse configurazioni di build su piattaforme Windows e Linux, evidenziando le differenze significative tra le modalità debug e release.

Piattaforma	Modalità	Componente	Dimensione (MB)	Note
Windows	Debug	ruggine-gui.exe	29.30	GUI Client con simboli debug
Windows	Debug	ruggine-server.exe	15.16	Server con simboli debug
Windows	Release	ruggine-gui.exe	13.63	GUI Client ottimizzato
Windows	Release	ruggine-server.exe	7.90	Server ottimizzato
Linux	Debug	ruggine-server	270.67	Server cross-compilato
Linux	Debug	ruggine-gui	296.26	GUI Client cross-compilato

Ottimizzazione Windows Release

Le build release Windows mostrano una riduzione del **53%** per il GUI client (da 29.30 MB a 13.63 MB) e del **48%** per il server (da 15.16 MB a 7.90 MB).

Dimensioni Linux Debug

I binari Linux debug sono notevolmente più grandi: **296.26 MB** per il client GUI e **270.67 MB** per il server, oltre 10 volte superiori rispetto a Windows.