Real-time Chat Application: Design Document

Overview

This document outlines the architecture for a real-time chat application using WebSockets. It solves the key architectural challenge of maintaining persistent, bi-directional communication channels between a server and multiple web clients, enabling instant message delivery, user presence, and conversation history. The design balances simplicity for learning with scalable patterns used in production systems.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This foundational context applies to all milestones, establishing the core communication paradigm and technical challenges the entire system must address.

At the heart of any real-time chat application lies a fundamental challenge: how can a web server instantly notify connected clients when something happens—like a new message being sent—without waiting for the client to ask first? This section explores this challenge by contrasting the traditional request-response model of the web with the persistent, bi-directional connection model of WebSockets. We will establish the mental models, define the core technical problems of managing stateful connections at scale, and evaluate the trade-offs between different real-time communication approaches.

Mental Model: The Telephone vs. The Mailbox

To understand the shift required for real-time communication, consider two everyday communication systems:

1. **The Mailbox (HTTP Request-Response):** Imagine you want to know if your friend has sent you a letter. Under the Hypertext Transfer Protocol (HTTP) that powers most of the web, you must walk to your mailbox, open it, and check for new letters. If there's nothing, you close it and return to your house. To get the next update, you must walk back to the mailbox and check again. This is the **polling** model: the client (you) must repeatedly initiate a request to the server (the mailbox) to ask for new data. The server cannot proactively deliver information; it can only respond when asked. This is efficient for infrequent updates but becomes wasteful and slow when you need to know the instant a letter arrives. You would be making constant, unnecessary trips to the mailbox.
2. **The Telephone Call (WebSocket Persistent Connection):** Now, imagine you and your friend have an open telephone line. Once the call is connected, either of you can speak at any time. When your friend has something to say, they just say it, and you hear it immediately. There's no need for you to repeatedly ask, "Do you have anything new to say?" This is the **WebSocket** model: after an initial handshake (dialing the number), a persistent, two-way communication channel is established. The server can "push" data to the client the moment it becomes available, and the client can send data to the server just as freely. This enables true real-time interactivity but comes with the overhead of maintaining an open connection for each client.

This mental model highlights the paradigm shift. Traditional web development is built around stateless, transactional interactions (the mailbox). Real-time chat requires a stateful, session-oriented, and continuous flow of information (the telephone call). Our design must therefore manage not just individual requests, but long-lived **connections** that become first-class citizens in our system's architecture.

The Core Problem: Stateful Connections at Scale

Building a system that maintains a "telephone call" with every user introduces a set of interconnected technical hurdles that define the core problem space for our chat application.

1. **Managing Thousands of Concurrent, Stateful Connections:** Unlike a stateless HTTP request that is processed and forgotten, each WebSocket connection is a long-lived object in server memory. It holds resources like file descriptors, buffers, and—critically—application state (e.g., which user it represents, which room they are in). The server must track all active connections, associate them with users, and efficiently route messages to the correct subset. As user count grows, the management overhead (memory, CPU for housekeeping) grows linearly. The system must be designed to handle this growth gracefully, typically using efficient data structures (like `Map`s keyed by user ID) and non-blocking, event-driven I/O.
2. **Handling Disconnections Gracefully and Reliably:** Network connections are inherently unreliable. A user's Wi-Fi may drop, their laptop may go to sleep, or they may close the browser tab. The server must detect when a connection is lost and perform necessary cleanup: removing the user from room member lists, broadcasting "user left" notifications, and freeing up allocated resources. Failure to do so creates **"ghost users"** (users who appear online but aren't) and memory leaks. This requires implementing a **connection lifecycle** with explicit events (`connect`, `disconnect`, `error`) and often a **heartbeat** mechanism (periodic ping/pong messages) to distinguish a slow connection from a dead one.
3. **Synchronizing State Across Clients in Real-Time:** When User A sends a message to "General" room, all other users in that room must see it nearly simultaneously. This is a state synchronization problem. The server acts as the single source of truth. It must:
    - **Receive** the message from User A's connection.
    - **Validate & Persist** it (check permissions, store to database if required).

    - **Determine Recipients** (find all connections currently in the "General" room, excluding User A).
    - **Transmit** the message to each recipient's connection. Any delay or failure in this pipeline breaks the illusion of a shared, synchronous space. The design must ensure this pipeline is fast, atomic where needed (e.g., persisting before broadcasting to ensure no message is lost if the server crashes mid-send), and resilient to partial failures of individual clients.
4. **Securing a Long-Lived, Bi-Directional Channel:** An open connection is a larger attack surface than a series of short requests. The server must authenticate the user when the connection is established (or very soon after) and then authorize every subsequent action (e.g., "can this user post to this room?"). This is distinct from HTTP, where each request can carry authentication headers. We must also guard against abuse via the persistent channel, such as a malicious client flooding the server with high-volume messages.

> The critical insight is that a WebSocket server is fundamentally a **stateful message router**. It maintains a live mapping of `{User Identity -> Network Connection -> Room Membership}` and uses this map to fan out messages from any single point to many others. This is a different architectural pattern than the stateless request-processing of a typical REST API.

Existing Approaches and Trade-offs

Before settling on WebSockets, it's essential to understand the landscape of techniques for achieving real-time communication on the web. Each has different capabilities, browser support, and trade-offs. The following table compares the most common approaches.

Approach	How It Works	Pros	Cons	Best For
Naive Polling	Client repeatedly sends HTTP requests (e.g., every 2 seconds) asking "Any new messages?"	Extremely simple to implement. Works with any HTTP server.	High latency (up to poll interval). High server/network overhead (many empty requests). Inefficient.	Simple prototypes where real-time is not critical and load is negligible.
Long Polling	Client sends a request; server holds it open until new data is available or a timeout occurs. Client immediately re-connects after receiving a response.	Reduces empty requests. Lower latency than naive polling. Still uses standard HTTP.	Complex server-side connection management. Head-of-line blocking (a hanging request can delay others). Still has HTTP overhead per message.	Systems where WebSockets are not available and moderate latency is acceptable.
Server-Sent Events (SSE)	Client opens a persistent HTTP connection; server can send "events" as text streams. **One-way (server → client only)**.	Standardized, automatic reconnection. Simple protocol (plain text). Efficient for server-push scenarios.	**No bi-directional communication** (client cannot send data over the same channel). Limited to text data. Not supported in older IE/Edge.	Live news feeds, stock tickers, notifications—any update stream where the client only listens.
WebSockets	Full-duplex, persistent connection established after an HTTP "Upgrade" handshake. Data flows both ways independently over the same TCP socket.	**True real-time, bi-directional communication**. Low overhead per message (light framing, no HTTP headers). Efficient for high-frequency updates.	More complex protocol to implement directly (though libraries abstract this). Requires stateful connection management on the server.	**Interactive applications: chat, collaborative editing, live gaming, trading terminals.**

> **Decision: Use WebSockets as the Core Transport**
>
> - **Context**: Our functional goal is a bi-directional, low-latency chat application where clients both send (messages, typing indicators) and receive (messages, presence updates) data in real-time.
> - **Options Considered**:
>   1. **Long Polling**: Could simulate real-time but introduces higher latency and complexity in request handling.
>   2. **Server-Sent Events (SSE)**: Excellent for server-to-client push, but would require a separate channel (e.g., standard HTTP POST) for client-to-server messages, complicating the architecture.
>   3. **WebSockets**: Provides a single, unified, bi-directional channel ideal for conversational data flow.
> - **Decision**: Implement the real-time data plane using the WebSocket protocol.
> - **Rationale**: WebSockets are the industry standard for full-duplex web communication. They provide the lowest latency and most efficient use of network resources for our use case, where events flow constantly in both directions. While they require managing stateful connections, this complexity is inherent to the problem and is a valuable learning objective.
> - **Consequences**: We must implement a WebSocket server capable of handling the connection lifecycle, message framing, and error handling. We will rely on the `ws` library (for Node.js) to handle the low-level protocol details, allowing us to focus on the application logic.

The choice of WebSockets sets the stage for our component design. In the following sections, we will decompose the problem of building a stateful message router into discrete, manageable components: the **Connection Manager** (Milestone 1), the **Message Broker & Room Manager** (Milestones 2 & 3), and the **Persistence & Authentication Service** (Milestone 4).

Goals and Non-Goals

> **Milestone(s):** All milestones (1-4). This section defines the fundamental requirements and constraints that guide the entire system's architecture and implementation, providing clear boundaries for what will and won't be built.

Functional Goals (What it MUST do)

These are the core capabilities the system must deliver—the features users will directly interact with and depend upon. Think of these as the **contract with the end user**: if any of these are missing, the application fails to fulfill its basic purpose.

Feature	Description	Why It Matters	Milestone
Real-time Messaging	Messages typed by one user must appear on other users' screens with minimal perceptible delay (typically < 200ms). This includes text content, sender identification, and accurate timestamps.	This is the fundamental value proposition of a chat application. Without real-time delivery, it becomes email. The experience should feel immediate and conversational.	2
Multiple Chat Rooms	Users must be able to create, join, and leave distinct conversation spaces (rooms/channels). Messages sent to a room must only be delivered to users currently in that room.	This organizes conversations by topic, project, or team, preventing a single chaotic stream. It's a foundational pattern for scaling conversations beyond direct messaging.	3
User Presence	The system must indicate which users are currently online/connected and which rooms they are actively participating in. Other users should see when someone joins or leaves a room or the entire application.	Presence transforms a messaging tool from a passive mailbox into an active social space. It provides critical context about who is available for conversation.	2, 4
Message History	When a user joins a room, they must see a configurable number of recent messages that were sent before they arrived. Messages must be persisted to survive server restarts.	Conversation continuity is essential. Users shouldn't lose context when they reconnect, and need to catch up on what they missed.	4
Typing Indicators	When a user begins composing a message, a visual indicator (e.g., "Alice is typing...") must be broadcast to other users in the same room. The indicator must clear after a timeout or when the message is sent.	This provides crucial feedback about the cadence of a conversation, reducing interruptions and signaling engagement.	2
User Authentication	Users must provide credentials (username/password) to establish their identity before participating in chats. The server must validate this identity for each WebSocket connection.	Authentication prevents impersonation, enables personalization, and is a prerequisite for proper message attribution and authorization.	4
Connection Resilience	The application must automatically attempt to re-establish the WebSocket connection if it is dropped due to network instability. In-flight messages should be queued and delivered upon reconnection.	Real-world networks are unreliable. A chat app that breaks on a spotty Wi-Fi signal is unusable. This goal is about graceful degradation.	1, 2

> **Key Insight:** These functional goals are **interdependent**. Presence relies on real-time connections. Message history requires persistence. Room-based messaging necessitates user authentication for proper attribution. The architecture must be designed holistically to support these interactions.

Detailed Breakdown of Core Features:

1. **Real-time Messaging Protocol:** Beyond simple delivery, the system must define a structured message format. Every message must include:
   - `sender`: The authenticated username of the originator.
   - `content`: The plain-text message body (with length limits).
   - `timestamp`: A server-assigned UTC timestamp (not client-provided, to prevent clock-skew issues).
   - `roomId`: The identifier of the destination room. This structured data is the lifeblood of the system, flowing from client to server to other clients.
2. **Room Management Logic:** The system must maintain a **room directory**. Key operations include:
   - **Creation:** A user can request a new room with a unique name. The server must sanitize the name (removing special characters) and check for conflicts.
   - **Joining:** A user can request to join an existing room. The server must add the user to the room's member list and notify existing members.
   - **Listing:** An endpoint (HTTP or via WebSocket) must provide a list of available rooms, including metadata like member count and activity timestamp.
   - **Cleanup:** The system should implement logic to archive or delete empty rooms after a period of inactivity to prevent resource leaks.
3. **Presence System Components:** Presence is more than just "online." It has multiple states:
   - **Global Online Status:** The user has an active WebSocket connection to the server.
   - **Room Membership:** The user is actively joined to one or more specific rooms.
   - **Typing Activity:** The user is currently composing a message in a specific room. The system must track these states and broadcast relevant state changes (e.g., `user_joined`, `user_left`, `user_typing`) to affected room members.

Non-Functional Goals (How it should behave)

These goals define the **quality attributes** of the system—how well it performs its functions. They are often constraints on the design and implementation. Think of these as the **contract with the system operator** (the developer running the server) and the **implied contract with the user regarding experience quality**.

Attribute	Target / Requirement	Architectural Implications
Low Latency	Message delivery from sender to receiver should typically be under 200 milliseconds on a stable network. The 95th percentile (P95) latency should not exceed 500ms.	Demands efficient message routing (O(1) room member lookups), non-blocking I/O, and minimal serialization/deserialization overhead. In-memory data structures for active sessions are essential.
Connection Resilience	The client must automatically attempt to reconnect after a disconnection, with exponential backoff (e.g., 1s, 2s, 4s, 8s... max 30s). Pending messages should be queued locally and sent upon reconnection.	Requires a stable client-side connection state machine and a server that can handle reconnections gracefully (re-associating the new socket with the existing user session).
Basic Security	1. **Authentication:** Credentials must be validated; passwords must be hashed (e.g., with bcrypt) before storage. 2. **Authorization:** Users can only send messages to rooms they have joined. 3. **Input Validation:** All incoming messages must be validated for type, size, and content (sanitize HTML). 4. **Transport:** Use WSS (WebSocket Secure) in production to encrypt traffic.	Influences the connection setup flow (authenticate *before* upgrading to WebSocket), adds validation layers in message handlers, and requires secure session management.
Debuggability & Operability	The system must produce structured logs for key events: connection lifecycle, message broadcasts, room operations, and errors. Logs should include relevant IDs (userId, roomId, socketId).	Dictates the use of a logging library and the careful instrumentation of core components. Logs are the primary window into the running system for developers.
Resource Efficiency	A single server instance should comfortably support thousands of concurrent connections on modest hardware. Memory usage should scale linearly with active connections and rooms.	Requires careful management of in-memory state (leak-free cleanup), efficient broadcast algorithms (avoiding $O(N^2)$ operations), and the use of event-driven, non-blocking APIs.
Functional Correctness	The system must guarantee: 1) **Message Ordering:** Messages from a single user to a single room are delivered to all others in the order they were sent. 2) **At-Least-Once Delivery:** A message sent is delivered to all intended online recipients, barring catastrophic failure. 3) **No Cross-Talk:** Messages are never delivered to users outside the intended room.	These are **system invariants**. They guide the design of the message broker and necessitate idempotent message handling where possible.

> **Architectural Principle:** Non-functional goals often involve **trade-offs**. For example, achieving low latency might involve keeping more data in memory (trading memory for speed). Our design prioritizes latency and simplicity for the learning context, accepting that this limits horizontal scalability—a conscious trade-off captured in our Non-Goals.

The "Why" Behind Non-Functional Goals:

- **Low Latency** is non-negotiable for conversational flow. Studies show delays over 200ms become perceptible and disruptive to turn-taking in dialogue.
- **Connection Resilience** acknowledges the reality of mobile networks and Wi-Fi handoffs. A chat app that doesn't handle this feels brittle and unreliable.
- **Basic Security** is the minimum bar for any application handling user-generated content and identities. While not implementing enterprise-grade security, we must avoid obvious vulnerabilities like plain-text passwords or SQL injection.
- **Debuggability** is critical for a learning project. When things go wrong (and they will), clear logs are the fastest path to understanding. This also teaches good operational practices.
- **Resource Efficiency** ensures the project remains runnable on a learner's laptop and demonstrates awareness of server-side constraints.

Explicit Non-Goals (What it does NOT do)

Defining what is **out of scope** is as important as defining what's in scope. It prevents scope creep, focuses effort on core learning objectives, and makes explicit the limitations of the current design. This is the **anti-contract**—a clear statement of what users and developers should not expect.

Feature / Capability	Reason for Exclusion	Implication / Workaround
Video/Voice Chat	Significantly increases complexity (codecs, streaming, NAT traversal) and is a separate domain from real-time text messaging.	Users must rely on external tools (e.g., Zoom, Discord) for voice/video communication.
File Sharing / Rich Media	Introduces concerns around storage, bandwidth, virus scanning, and content moderation. Moves focus from message routing to file handling.	Users can share links to files stored elsewhere (e.g., Google Drive). The chat remains text-centric.
Advanced Moderation Tools	Tools like automated profanity filters, user banning, message deletion, or admin panels require a complex permissions system and content policy.	Room creators have no special powers. All users in a room are equal peers.
Horizontal Scaling (Multi-Server)	Distributing WebSocket connections and real-time state across multiple servers introduces massive complexity (sticky sessions, shared state via Redis/PubSub, distributed consensus).	The application is designed to run on a **single server process**. To scale, you would vertically scale (bigger machine) until this becomes limiting, then require a significant architectural rewrite.
Permanent Message Archive & Search	Full-text search across all historical messages requires dedicated search infrastructure (like Elasticsearch) and complex pagination/querying.	History is limited to a recent, configurable window (e.g., last 1000 messages per room). Searching is manual (Ctrl+F in the client).
Direct/Private Messaging (DMs)	While related to rooms, DMs require a different addressing scheme (user-to-user), privacy guarantees, and potentially a separate inbox model. It expands the core room-based mental model.	Users can create a private room named for two people, but there is no first-class "DM" concept or privacy enforcement.
Read Receipts ("Seen" indicators)	Requires tracking which messages have been *rendered* by each recipient, which is a client-side event that must be reported back to the server and stored. Adds significant state complexity.	Users only know a message was *delivered* (recipient was online and in the room), not necessarily *read*.
Message Editing or Deletion	Allowing edits/deletes after broadcast requires tracking message versions, propagating updates, and handling the "temporal consistency" problem (what if someone saw the original?).	Messages are immutable once sent. To "correct" a message, a user must send a new one.
Offline Message Queue & Push Notifications	Storing messages for offline users and delivering them via mobile push (APNS/FCM) requires a separate queueing system, device token management, and a significant service integration.	Messages are only delivered to users who are **currently online and joined to the room**. If you're offline, you miss the conversation until you reconnect and load history.
End-to-End Encryption (E2EE)	E2EE requires key exchange and management (e.g., Signal protocol), client-side encryption, and prevents the server from reading messages. This is a deep specialization in cryptographic engineering.	All messages are visible to the server (necessary for broadcasting and persistence). Transport is encrypted (WSS), but server operators can technically view all content.

> **Design Philosophy:** The choices in this table are guided by a **pedagogical focus**. We are building a **learning vehicle** for WebSockets, real-time state management, and basic full-stack architecture. Adding video chat or horizontal scaling would obscure these core lessons with ancillary complexity. This system is a **foundation**—a complete, working prototype that demonstrates the essential patterns. The "Future Extensions" section will discuss how one might evolve this foundation to support some of these excluded features.

The Consequences of Our Non-Goals: By explicitly rejecting horizontal scaling, we are free to use simple, in-memory data structures (`Map` and `Set`) for tracking connections and rooms. This makes the code dramatically easier to understand and debug. By rejecting offline messaging, we avoid building a persistent queue system. By rejecting file sharing, we don't need to integrate with object storage or handle multipart uploads. **These exclusions are liberating constraints that allow us to build a clean, understandable system focused on the core learning objectives.**

Implementation Guidance (Layer 2)

This section bridges the design concepts to concrete implementation choices and starter code.

A. Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
WebSocket Library	`ws` (minimal, follows standard WebSocket API)	`Socket.IO` (built-in reconnection, rooms, fallback to polling)
Server Framework	Native Node.js `http` / `https` modules	`Express.js` with middleware for HTTP routes
Authentication Storage	In-memory `Map` (for simplicity) or a simple JSON file	`SQLite` or `PostgreSQL` with a proper `users` table
Message Persistence	In-memory array per room (volatile) or append to a JSONL (JSON Lines) file	`SQLite` / `PostgreSQL` with a `messages` table and indexes on `roomId` and `timestamp`
Client Library	Native `WebSocket` API	`Socket.IO` client for automatic reconnection

> **Recommendation for Learners:** Start with the **Simple Options** (`ws` , native `http` , in-memory storage). They have fewer abstractions, making it easier to see how the underlying protocols work. You can graduate to the advanced options as a later exercise.

B. Recommended File/Module Structure: Organizing code from the start prevents a monolithic `server.js` file. Here's a suggested structure aligning with our components.

```
real-time-chat/
├── package.json
├── server/
│   ├── index.js                    # Main entry point, creates HTTP server
│   ├── connectionManager/          # Milestone 1: WebSocket Server & Connection Manager
│   │   ├── index.js                # Exports the ConnectionManager class
│   │   └── heartbeat.js            # Ping/pong heartbeat logic
│   ├── roomManager/                # Milestones 2 & 3: Message Broker & Room Manager
│   │   ├── index.js                # Exports the RoomManager class
│   │   └── typingIndicator.js      # Logic for managing typing timeouts
│   ├── persistence/                # Milestone 4: Persistence & Authentication
│   │   ├── authService.js          # User registration, login, session validation
│   │   ├── messageStore.js         # Save and load messages from storage
│   │   └── simpleStore.js          # In-memory or file-based store (starter)
│   └── utils/
│       ├── logger.js               # Structured logging utility
│       ├── validation.js           # Input validation helpers
│       └── constants.js            # Shared constants (event names, limits)
└── client/
    ├── public/
    │   ├── index.html              # Main chat UI
    │   ├── style.css               # Styles
    │   └── app.js                  # Client-side WebSocket & UI logic
```

C. Infrastructure Starter Code (Simple In-Memory Store): This is a complete, working `simpleStore.js` you can use for early milestones before adding a database. It provides a basic key-value interface.

JAVASCRIPT

```javascript
// server/persistence/simpleStore.js

/**
 * A simple, in-memory key-value store with optional file persistence.
 * NOT for production use - data is lost on server restart unless saveToFile is used.
 */
class SimpleStore {
  constructor(persistFile = null) {
    this.data = new Map(); // key -> value
    this.persistFile = persistFile;
    this.loadFromFile();
  }

  // Load initial data from a JSON file
  loadFromFile() {
    if (!this.persistFile) return;
    try {
      const fs = require('fs');
      if (fs.existsSync(this.persistFile)) {
        const raw = fs.readFileSync(this.persistFile, 'utf8');
        const entries = JSON.parse(raw);
        this.data = new Map(entries);
        console.log(`Loaded ${this.data.size} entries from ${this.persistFile}`);
      }
    } catch (err) {
      console.error('Failed to load persistent store:', err);
    }
  }

  // Save current state to a JSON file
  saveToFile() {
    if (!this.persistFile) return;
    try {
      const fs = require('fs');
      const entries = Array.from(this.data.entries());
      const raw = JSON.stringify(entries, null, 2);
      fs.writeFileSync(this.persistFile, raw, 'utf8');
    } catch (err) {
      console.error('Failed to save persistent store:', err);
    }
  }

  // Core CRUD operations
```

```
  set(key, value) {
    this.data.set(key, value);
    this.saveToFile();
    return true;
  }

  get(key) {
    return this.data.get(key);
  }

  has(key) {
    return this.data.has(key);
  }

  delete(key) {
    const result = this.data.delete(key);
    this.saveToFile();
    return result;
  }

  // Find all keys where the value matches a predicate
  find(predicate) {
    const results = [];
    for (const [key, value] of this.data.entries()) {
      if (predicate(value, key)) {
        results.push({ key, value });
      }
    }
    return results;
  }

  // Get all entries as an array
  getAll() {
    return Array.from(this.data.entries());
  }
}

module.exports = SimpleStore;
```

D. Core Logic Skeleton Code (Connection Manager): Here is the skeleton for the main `ConnectionManager` class. Fill in the TODOs based on the algorithm steps described in the Component Design section.

JAVASCRIPT

```javascript
// server/connectionManager/index.js

const WebSocket = require('ws');

const logger = require('../utils/logger');

const { validateMessage } = require('../utils/validation');


class ConnectionManager {

  constructor(server) {

    // ADR: Using Map for direct socket access by a unique connectionId

    this.clients = new Map(); // connectionId -> { socket, userId, ... }

    this.wss = new WebSocket.Server({ server });


    this.setupEventHandlers();

    this.startCleanupInterval();

  }


  setupEventHandlers() {

    this.wss.on('connection', (socket, request) => {

      // TODO 1: Generate a unique connectionId (e.g., uuid or timestamp+random)

      // TODO 2: Create a client object with socket, connectionId, and metadata (ip, userAgent)

      // TODO 3: Store client in this.clients Map

      // TODO 4: Attach message, close, and error event handlers to the socket

      // TODO 5: Log the connection event

      // TODO 6: Optional: Implement immediate authentication check here (see Milestone 4 ADR)

    });

  }


  // Handle an incoming message from a client

  handleMessage(socket, rawMessage) {

    // TODO 1: Try to parse rawMessage as JSON, catch error and send error response

    // TODO 2: Validate the parsed message has required fields (type, payload)

    // TODO 3: Look up the client object associated with this socket

    // TODO 4: Route the message to the appropriate handler (e.g., roomManager.broadcast)

    // TODO 5: Wrap in try/catch - log any handler error and send error response to client

  }


  // Handle client disconnection

  handleClose(socket, code, reason) {

    // TODO 1: Find the client object for this socket

    // TODO 2: If found, notify the roomManager that this user left all rooms

    // TODO 3: Remove the client from this.clients Map

    // TODO 4: Log the disconnection event

  }
```

```
  // Send a message to a specific client socket
  sendToClient(connectionId, message) {
    // TODO 1: Look up client by connectionId
    // TODO 2: Check if client.socket.readyState === WebSocket.OPEN
    // TODO 3: If open, stringify message and send via client.socket.send()
    // TODO 4: If not open, log a warning and optionally clean up the client
  }

  // Broadcast a message to multiple clients
  broadcastToClients(connectionIds, message) {
    // TODO 1: Iterate over connectionIds array
    // TODO 2: For each id, call this.sendToClient(id, message)
    // TODO 3: Consider batching or using Promise.all if performance becomes an issue
  }

  // Periodic cleanup of dead connections (fail-safe)
  startCleanupInterval() {
    setInterval(() => {
      // TODO 1: Iterate over this.clients
      // TODO 2: For each client, check if socket.readyState is CLOSED or CLOSING
      // TODO 3: If so, call this.handleClose for that socket
    }, 30000); // Run every 30 seconds
  }

  // Get statistics (for debugging/admin)
  getStats() {
    return {
      totalConnections: this.clients.size,
      // TODO: Add more stats like connections per user, etc.
    };
  }
}

module.exports = ConnectionManager;
```

E. Language-Specific Hints (JavaScript/Node.js):

- **Use `ws` library:** Install via `npm install ws`. It's the most standard WebSocket server implementation for Node.js.
- **Connection IDs:** Use `crypto.randomUUID()` (Node 15+) or the `uuid` package to generate unique connection identifiers.
- **Heartbeats:** Implement using `socket.ping()` and `socket.pong()` listeners. Set a timer; if no pong is received within a timeout, terminate the connection.
- **Error Handling:** Always attach an `'error'` listener to the WebSocket socket. Unhandled socket errors can crash the Node.js process.
- **Logging:** Use a simple wrapper around `console.log` that prefixes with timestamps and log levels (INFO, ERROR). This dramatically improves debuggability.

F. Milestone Checkpoint (End of Milestone 1): After implementing the Connection Manager skeleton:

1. **Run the server:** `node server/index.js`

2. **Test connection:** Open `client/public/index.html` in a browser (or use a WebSocket testing tool like `wscat`). You should see "Connected to server" in your server logs.
3. **Test multiple clients:** Open a second browser tab. Both should connect independently.
4. **Test disconnection:** Close one tab. Within a few seconds, you should see a "Client disconnected" log.
5. **Verify cleanup:** Check that your `this.clients.size` decreases after a disconnection (you can add a temporary log or admin endpoint).
6. **Expected Signs of Trouble:**
   - **Server crashes on connect/disconnect:** Likely an unhandled error event on the socket. Add error listeners everywhere.
   - **Client count keeps increasing:** You are not removing clients on disconnect. Check your `handleClose` logic and cleanup interval.
   - **No logs appear:** Ensure your logger is called at each lifecycle event.

High-Level Architecture

> **Milestone(s):** This architectural overview provides the foundation for all four milestones, outlining the major components and their interactions that will be built and integrated throughout the project.

The high-level architecture visualizes the system as four collaborating pillars, each with distinct responsibilities but working together to deliver the chat experience. This decomposition follows the **separation of concerns** principle: each component handles one core aspect of the system, making the codebase easier to understand, test, and maintain.

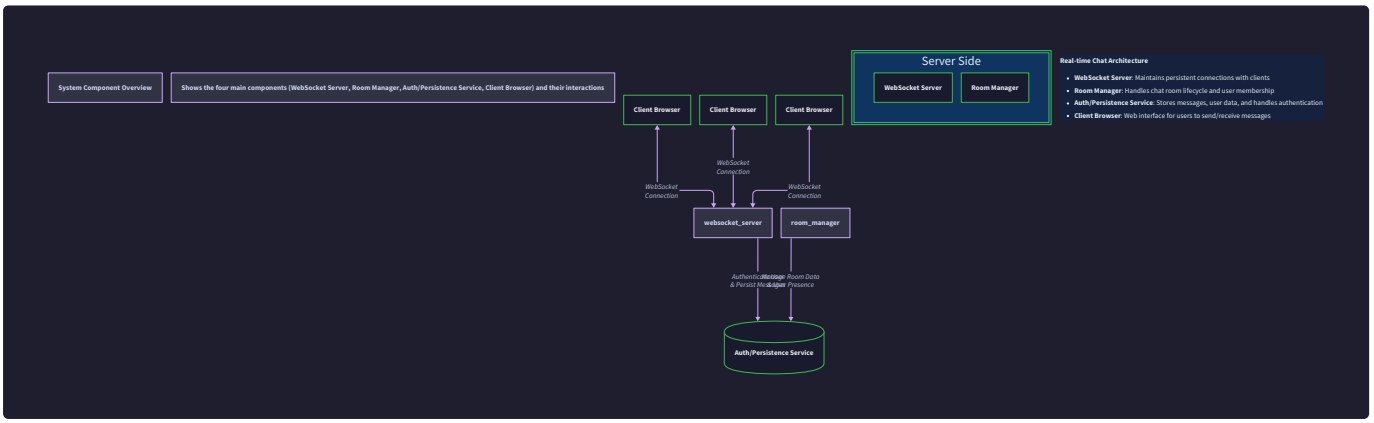
Component Overview and Responsibilities

Think of the chat system as a **modern conference center**:

- **The WebSocket Server** is the reception desk and switchboard, greeting visitors and connecting their calls
- **The Room & Presence Manager** is the conference coordinator, tracking who's in which meeting room and what they're doing
- **The Client Application** is the attendee's smartphone app, providing the interface to participate
- **The Persistence Layer** is the archives and security office, storing historical records and verifying identities

This mental model helps visualize how responsibilities are divided while maintaining clear communication channels between components.

The following diagram illustrates the component architecture and their primary interactions:



Core Components and Their Responsibilities

Component	Primary Responsibility	Key Data Owned	Critical Operations	Interfaces With
WebSocket Server & Connection Manager	Manages the lifecycle of real-time connections	Set of active `ClientSession` objects, socket state	Accept connections, route messages, handle disconnections	Client browsers (via WebSocket), Room Manager, Auth Service
Room & Presence Manager	Orchestrates room-based messaging and user presence	Collection of `Room` objects with member lists, typing status	Add/remove users from rooms, broadcast messages, track presence	WebSocket Server (for message delivery), Persistence Layer (for history)
Persistence & Authentication Service	Stores message history and manages user identity	`User` records, `ChatMessage` history in database	User registration/login, message storage/retrieval, session validation	WebSocket Server (for auth), Room Manager (for history)
Client Application	Provides the user interface for chat interaction	Local message cache, UI state, connection status	Render messages, capture user input, manage WebSocket connection	WebSocket Server (via WebSocket protocol)

Detailed Component Descriptions

WebSocket Server & Connection Manager This component serves as the **system's entry point** for all real-time communication. It's responsible for the low-level WebSocket protocol handling: accepting HTTP upgrade requests, establishing persistent connections, managing connection state, and providing a clean interface for other components to send messages to clients. It maintains a live registry of all connected clients as `ClientSession` objects, each representing a single WebSocket connection with associated metadata (user ID, IP address, user agent). The server implements **heartbeat mechanisms** (ping/pong) to detect stale connections and ensures proper cleanup when clients disconnect.

Room & Presence Manager Operating as the **application's business logic layer**, this component implements the chat-specific functionality. It manages the concept of rooms (chat channels), tracks which users are in which rooms, handles message broadcasting with appropriate exclusions, and maintains presence information (online/offline status, typing indicators). When a message arrives from a client, the Room Manager determines which room it belongs to, retrieves the list of members in that room (excluding the sender), and instructs the WebSocket Server to deliver the message to each member's socket. It also handles room lifecycle—creating rooms when the first user joins and cleaning up empty rooms if configured to do so.

Persistence & Authentication Service This dual-purpose component provides **data durability and security**. The authentication portion handles user registration, login credential verification, and session management. When a WebSocket connection attempt occurs, this service validates the provided credentials (typically via token or cookie) before allowing the upgrade. The persistence portion stores all chat messages in a database with proper indexing by room and timestamp, enabling historical message retrieval. It also manages user profiles and, in more advanced implementations, could handle read receipts or message editing history.

Client Application The client is a **stateful web application** running in the user's browser. It establishes and maintains the WebSocket connection to the server, handles connection retries with exponential backoff, renders the chat interface, and manages local UI state (current room, message draft, unread counts). The client implements the event protocol defined by the server, sending properly formatted JSON messages for chat events and handling incoming messages to update the UI in real-time.

Component Interaction Patterns

The components interact through three primary patterns:

1. **Request-Response over WebSocket**: The client sends a message (like `join_room`), the server processes it through the appropriate handlers, and sends back a response (like `room_joined` with message history).
2. **Event Notification**: When something happens that other clients should know about (like a user joining a room), the Room Manager instructs the WebSocket Server to broadcast an event (like `EVENT_USER_JOINED`) to all relevant clients.
3. **Data Access**: The Room Manager queries the Persistence Layer for message history when a user joins a room, and writes new messages to it after validation.

> **Key Design Insight:** The WebSocket Server acts as a **message transport layer**, while the Room Manager contains the **application logic**. This separation allows the transport mechanism to potentially be swapped (e.g., for a different real-time protocol) without rewriting the core chat functionality.

Data Flow Between Components

When a user sends a chat message, the data flows through the system as follows:

1. **Client** → **WebSocket Server**: Raw WebSocket frame containing JSON
2. **WebSocket Server** → **Room Manager**: Parsed message object with sender metadata
3. **Room Manager** → **Persistence Layer**: Message stored to database
4. **Room Manager** → **WebSocket Server**: List of recipient connection IDs and formatted message
5. **WebSocket Server** → **Multiple Clients**: Individual WebSocket frames to each recipient

This flow ensures that business logic (room membership, message formatting) remains separate from transport concerns (socket management, connection state).

Recommended File/Module Structure

Organizing the codebase with a clear structure from the beginning prevents the common "everything in one file" antipattern that makes projects difficult to maintain. The recommended structure follows the **component boundaries** established in the architecture, with clear separation between transport logic, business logic, persistence, and client code.

```
real-time-chat/
├── server/                        # Server-side code
│   ├── src/                       # TypeScript source
│   │   ├── core/                  # Foundational utilities
│   │   │   ├── logger.ts          # Structured logging
│   │   │   ├── config.ts          # Configuration management
│   │   │   └── errors.ts          # Custom error classes
│   │   ├── transport/             # WebSocket Server component
│   │   │   ├── websocket-server.ts  # Main WebSocket server class
│   │   │   ├── connection-manager.ts # Manages ClientSession objects
│   │   │   ├── protocol/          # Message protocol definitions
│   │   │   │   ├── message-types.ts # Constants like EVENT_CHAT_MESSAGE
│   │   │   │   ├── validators.ts    # Message validation functions
│   │   │   │   └── serializers.ts   # Message formatting utilities
│   │   │   └── middleware/        # WebSocket middleware
│   │   │       ├── auth-middleware.ts # Authentication on upgrade
│   │   │       └── heartbeat-middleware.ts # Ping/pong handling
│   │   ├── rooms/                 # Room & Presence Manager component
│   │   │   ├── room-manager.ts      # Main room management class
│   │   │   ├── presence-tracker.ts  # Tracks online/typing status
│   │   │   ├── room.ts              # Room class definition
│   │   │   └── types.ts             # TypeScript interfaces for rooms
│   │   ├── persistence/           # Persistence & Auth component
│   │   │   ├── auth/              # Authentication subsystem
│   │   │   │   ├── auth-service.ts  # User registration/login
│   │   │   │   ├── session-store.ts # Manages active sessions
│   │   │   │   └── password-utils.ts # Password hashing/verification
│   │   │   ├── database/          # Database abstraction
│   │   │   │   ├── message-store.ts # ChatMessage CRUD operations
│   │   │   │   ├── user-store.ts    # User profile management
│   │   │   │   └── base-repository.ts # Shared database logic
│   │   │   └── models/            # Data model definitions
│   │   │       ├── chat-message.ts  # ChatMessage type/interface
│   │   │       ├── user.ts          # User type/interface
│   │   │       └── client-session.ts # ClientSession type/interface
│   │   └── main.ts                # Application entry point
│   ├── tests/                     # Server-side tests
│   │   ├── unit/                  # Unit tests by component
│   │   └── integration/           # Integration tests
│   ├── package.json               # Node.js dependencies
│   └── tsconfig.json              # TypeScript configuration
├── client/                        # Web client application
│   ├── src/                       # Client source code
│   │   ├── lib/                   # Client-side libraries
│   │   │   └── websocket-client.ts  # WebSocket connection wrapper
│   │   ├── ui/                    # User interface components
│   │   │   ├── chat-room.ts         # Main chat room component
│   │   │   ├── message-list.ts      # Message display component
│   │   │   ├── user-list.ts         # Online users component
│   │   │   └── message-input.ts     # Message composition component
│   │   ├── state/                 # Client-side state management
│   │   │   ├── connection-store.ts  # WebSocket connection state
│   │   │   ├── message-store.ts     # Local message cache
│   │   │   └── user-store.ts        # Local user/profile state
│   │   ├── utils/                 # Client utilities
│   │   │   └── message-formatter.ts # Format messages for display
│   │   └── main.ts                # Client application entry point
│   ├── public/                    # Static assets
│   │   └── index.html             # Main HTML file
│   └── package.json               # Client dependencies
├── shared/                        # Code shared between client and server
│   └── types/                     # Shared TypeScript definitions
│       └── protocol.ts            # Common message type definitions
└── docker-compose.yml             # Local development with database
```

Key Structural Decisions

Decision: Separating Transport from Business Logic

- **Context**: The system needs to handle both the low-level WebSocket protocol and high-level chat semantics
- **Options Considered**:
  1. Combined transport and business logic in one module
  2. Separate transport layer with clean interface to business logic
- **Decision**: Separate transport (`transport/`) from business logic (`rooms/`)
- **Rationale**: This separation allows each to evolve independently—the transport could switch to a different protocol (like Socket.IO or raw TCP) without affecting room management logic. It also enables cleaner testing, as the room logic can be tested without actual WebSocket connections.

- **Consequences**: Requires well-defined interfaces between layers, slightly more code organization overhead, but yields more maintainable and testable code.

Decision: Client-Server Shared Types

- **Context**: Both client and server need to agree on message formats and data structures
- **Options Considered**:
  1. Duplicate type definitions in both codebases
  2. Shared TypeScript definitions in a common directory
- **Decision**: Shared type definitions in `shared/types/`
- **Rationale**: This ensures type safety across the entire stack—if a message format changes, TypeScript will catch inconsistencies at compile time rather than runtime. It's particularly valuable for the event protocol where client and server must agree exactly on field names and types.
- **Consequences**: Requires build configuration to include shared directory in both client and server TypeScript compilation, but prevents a whole class of protocol mismatch bugs.

Decision: Database Abstraction Layer

- **Context**: The system needs to store messages and user data, but the specific database technology might evolve
- **Options Considered**:
  1. Direct database calls throughout the codebase
  2. Repository pattern with clean interfaces
- **Decision**: Repository pattern in `persistence/database/`
- **Rationale**: This abstraction allows switching database technologies (from SQLite for development to PostgreSQL for production) with minimal code changes. It also centralizes database queries, making optimization and debugging easier.
- **Consequences**: Adds an extra layer of abstraction but significantly improves long-term maintainability and testability (via mocking).

> **Implementation Note:** For simplicity in early milestones, you may start with in-memory storage in the Room Manager and add persistence later. The architecture supports this progression—the Room Manager interface to the persistence layer remains the same whether it's talking to an in-memory store or a real database.

Module Dependencies and Import Flow

The dependency flow follows a **unidirectional architecture**:

```
main.ts → websocket-server.ts → connection-manager.ts
                              ↗
room-manager.ts ← persistence/ (auth-service, message-store)
```

Key dependency rules:

1. **High-level components depend on low-level ones**, not vice versa
2. **The `transport/` layer knows about `rooms/` and `persistence/`** for message routing
3. **The `rooms/` layer knows about `persistence/`** for data access
4. **The `persistence/` layer has no dependencies** on other business logic components
5. **Shared types have no dependencies** and are imported by both client and server

This structure prevents circular dependencies and makes the system easier to reason about.

Implementation Guidance

> **Technology Note:** For this JavaScript/TypeScript project, we recommend starting with Node.js and the `ws` library for WebSocket support, as it provides a minimal, standards-compliant implementation without the additional abstractions of Socket.IO. This approach gives you direct exposure to the WebSocket protocol fundamentals.

Technology Recommendations

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
WebSocket Server	Node.js + `ws` library	Node.js + `ws` with connection pooling
Room Management	In-memory JavaScript objects	Redis for distributed room state
Persistence	SQLite with `better-sqlite3`	PostgreSQL with connection pooling
Authentication	JWT tokens stored in memory	OAuth2.0 with persistent session store
Client Framework	Vanilla JavaScript + DOM API	React/Vue.js with state management

Starter Project Structure Setup

Create the basic project structure with these commands:

```bash
# Create project root
mkdir real-time-chat
cd real-time-chat

# Initialize server
mkdir -p server/src/{core,transport/{protocol,middleware},rooms,persistence/{auth,database,models}}
mkdir server/tests/{unit,integration}
cd server
npm init -y
npm install ws typescript @types/node @types/ws
npm install -D ts-node nodemon jest @types/jest

# Initialize client
cd ..
mkdir -p client/src/{lib,ui,state,utils}
mkdir client/public
cd client
npm init -y
npm install typescript

# Create shared types directory
cd ..
mkdir -p shared/types

# Create TypeScript configurations
# (Details in next section)
```

Basic TypeScript Configuration

Create `server/tsconfig.json`:

```json
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "lib": ["ES2020"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "types": ["node", "jest"]
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist", "tests"]
}
```

Create `client/tsconfig.json`:

```json
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "lib": ["ES2020", "DOM"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "moduleResolution": "node",
    "resolveJsonModule": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Core Infrastructure Starter Code

Shared Protocol Types (`shared/types/protocol.ts`):

TYPESCRIPT

```typescript
// Shared type definitions between client and server

export interface ChatMessage {

  sender: string;

  content: string;

  timestamp: string; // ISO format

  roomId: string;

}


export interface User {

  username: string;

  passwordHash: string;

  createdAt: Date;

}


export interface ClientSession {

  connectionId: string;

  socket: WebSocket; // Note: WebSocket type differs client vs server

  userId: string;

  ip: string;

  userAgent: string;

}


// Event type constants

export const EVENT_CHAT_MESSAGE = 'chat_message';

export const EVENT_USER_JOINED = 'user_joined';

export const EVENT_USER_TYPING = 'user_typing';

export const MAX_MESSAGE_LENGTH = 1000;


// Message envelope for WebSocket communication

export interface WebSocketMessage {

  type: string;

  payload: any;

  timestamp?: string;

}
```

Server Core Utilities (`server/src/core/logger.ts`):

```typescript
// Simple structured logger for development
export class Logger {
  static info(message: string, metadata: any = {}) {
    console.log(JSON.stringify({
      level: 'INFO',
      timestamp: new Date().toISOString(),
      message,
      ...metadata
    }));
  }

  static error(message: string, error: any = {}, metadata: any = {}) {
    console.error(JSON.stringify({
      level: 'ERROR',
      timestamp: new Date().toISOString(),
      message,
      error: error.message || String(error),
      stack: error.stack,
      ...metadata
    }));
  }

  static warn(message: string, metadata: any = {}) {
    console.warn(JSON.stringify({
      level: 'WARN',
      timestamp: new Date().toISOString(),
      message,
      ...metadata
    }));
  }
}
```

TYPESCRIPT

Server Entry Point Skeleton (`server/src/main.ts`):

```typescript
import { WebSocketServer } from './transport/websocket-server';
import { Logger } from './core/logger';

const PORT = process.env.PORT || 8080;

async function main() {
  Logger.info('Starting real-time chat server', { port: PORT });

  // TODO: Initialize database connection if using persistence


  // Create and start WebSocket server
  const server = new WebSocketServer({
    port: Number(PORT),
    // TODO: Pass configuration for room manager, auth service, etc.
  });

  server.start();

  // Graceful shutdown handling
  process.on('SIGTERM', () => {
    Logger.info('Received SIGTERM, shutting down gracefully');
    server.stop();
    process.exit(0);
  });

  process.on('SIGINT', () => {
    Logger.info('Received SIGINT, shutting down gracefully');
    server.stop();
    process.exit(0);
  });
}

main().catch((error) => {
  Logger.error('Failed to start server', error);
  process.exit(1);
});
```

Client WebSocket Wrapper Skeleton (`client/src/lib/websocket-client.ts`):

```typescript
import { WebSocketMessage } from '../../../shared/types/protocol';

export class WebSocketClient {
  private socket: WebSocket | null = null;
  private reconnectAttempts = 0;
  private maxReconnectAttempts = 5;
  private reconnectDelay = 1000;

  constructor(private url: string) {}

  connect(): void {
    // TODO 1: Create new WebSocket connection to this.url
    // TODO 2: Set up event handlers for open, message, close, error
    // TODO 3: Implement exponential backoff for reconnection
    // TODO 4: Store connection state for UI feedback
  }

  send(message: WebSocketMessage): void {
    // TODO 1: Check if socket is in OPEN state (readyState === 1)
    // TODO 2: Stringify message to JSON
    // TODO 3: Send via socket.send()
    // TODO 4: Handle errors (queue for retry or notify UI)
  }

  disconnect(): void {
    // TODO 1: Close WebSocket connection gracefully
    // TODO 2: Clean up event listeners
    // TODO 3: Reset reconnect state
  }

  // TODO: Add methods for specific message types (sendChatMessage, joinRoom, etc.)
}
```

Development Scripts

Add these scripts to `server/package.json`:

```json
{
  "scripts": {
    "build": "tsc",
    "start": "node dist/main.js",
    "dev": "nodemon --exec ts-node src/main.ts",
    "test": "jest",
    "test:watch": "jest --watch"
  }
}
```

Next Steps for Implementation

With this architecture in place, you'll implement the system component by component:

1. **Start with the WebSocket Server** (Milestone 1): Implement the connection lifecycle in `transport/`
2. **Add Room Management** (Milestones 2-3): Build the room logic in `rooms/`
3. **Implement Persistence** (Milestone 4): Add database storage in `persistence/`
4. **Build the Client** (Parallel development): Create the UI in `client/`

The clear separation between components allows you to work on one layer at a time, testing each in isolation before integrating them into the complete system.

Data Model

> **Milestone(s):** All milestones (1-4). This section defines the foundational data structures and communication protocol that enable real-time messaging, room management, user presence, and message persistence across the entire system. The data model serves as the contract between components and clients.

Core Types and Structures

Think of the data model as the **DNA of your chat application**—it encodes all the information needed to represent users, conversations, and interactions. Just as DNA has four nucleotide bases (A, T, C, G) that combine in specific patterns to create genetic instructions, your chat system has four core data types (`ChatMessage`, `User`, `Room`, and `ClientSession`) that combine to represent the complete state of the application. These structures flow through your system like genetic information flows through a biological organism, providing the blueprint for all functionality.

The following tables define each core type with exact field specifications. Note that these are **in-memory representations** used within the server—the wire format for transmitting this data to clients may differ slightly (e.g., using ISO string timestamps instead of Date objects).

ChatMessage

A `ChatMessage` represents a single unit of communication between users. It's the atomic building block of conversation—like a sentence in a dialogue. Each message is immutable once created and serves as a historical record of what was said, by whom, when, and where.

Field Name	Type	Description & Constraints
`sender`	`string`	The username of the user who sent this message. This must match an existing user's username in the system. Cannot be empty or contain only whitespace.
`content`	`string`	The textual content of the message. Maximum length is defined by `MAX_MESSAGE_LENGTH` (1000 characters). Must be validated for length and sanitized for potentially malicious content (e.g., HTML tags) before storage or transmission.
`timestamp`	`string` (ISO 8601)	The UTC date and time when the server received and processed the message, expressed in ISO 8601 format (e.g., `"2024-01-15T10:30:45.123Z"`). Using a string format ensures consistent parsing across different clients and languages. The server assigns this timestamp, not the client, to prevent clock skew issues.
`roomId`	`string`	The unique identifier of the room where this message was sent. This must correspond to an existing room's `id` field. Messages are always scoped to a specific room—they cannot exist outside of a room context.

> **Design Insight:** The `timestamp` field uses ISO 8601 string format rather than a numeric Unix timestamp for better human readability during debugging and compatibility with JavaScript's `Date.parse()` and `toISOString()` methods. However, this comes with a slight storage overhead compared to storing milliseconds as a number.

User

A `User` represents a registered person who can participate in chat rooms. Think of it as a **citizen of your chat universe**—they have an identity (`username`), proof of citizenship (`passwordHash`), and a record of when they joined the community (`createdAt`). The `User` object contains authentication credentials and metadata, but note that it does NOT contain connection state—that's tracked separately in `ClientSession`.

Field Name	Type	Description & Constraints
`username`	`string`	Unique identifier for the user across the system. Typically 3-20 characters, alphanumeric with optional underscores or hyphens. Case-sensitive for simplicity (though you could normalize to lowercase).
`passwordHash`	`string`	Hashed representation of the user's password using a secure one-way hash function (e.g., bcrypt, Argon2). Never store passwords in plain text. The hash should include a salt to prevent rainbow table attacks.
`createdAt`	`Date`	The date and time when the user account was created. Used for administrative purposes and potentially for displaying "member since" information. Stored as a Date object server-side but serialized to ISO string when transmitted.

> **Design Insight:** The `User` object is separate from connection state (`ClientSession`) because a single user can be connected from multiple devices simultaneously (e.g., phone and laptop). This separation enables features like "last seen" timestamps and multi-device message synchronization in more advanced implementations.

Room

A `Room` represents a named conversation space where users gather to exchange messages. Imagine it as a **virtual conference room** with a door, a nameplate, and a list of who's currently inside. Rooms are dynamic containers that hold both members (via `memberIds`) and the history of their conversation (via associated `ChatMessage` records in the database).

Field Name	Type	Description & Constraints
`id`	`string`	Unique identifier for the room, typically generated as a UUID or a URL-safe version of the room name. Used as a stable reference in URLs and message routing.
`name`	`string`	Human-readable room name (e.g., "general", "random", "project-discussion"). Displayed in room lists and UI headers. Should be sanitized to prevent injection attacks.
`createdAt`	`Date`	When the room was created. Useful for administrative cleanup of old, inactive rooms.
`memberIds`	`Set<string>`	A collection of user IDs (usernames) who are currently present in the room. Using a `Set` ensures each user appears only once and provides O(1) membership tests. This set is updated in real-time as users join and leave.

> **Design Insight:** The `memberIds` field uses a `Set` rather than an `Array` for efficient membership testing (`Set.has()` is O(1) vs `Array.includes()` which is O(n)). This performance difference becomes critical when broadcasting messages to rooms with hundreds of members.

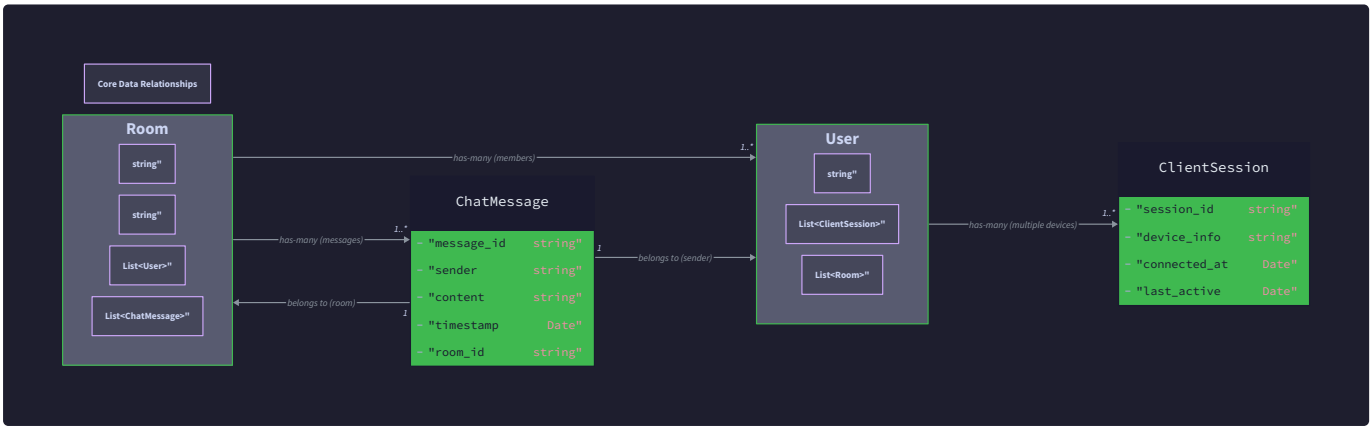
ClientSession

A `ClientSession` represents an **active WebSocket connection** from a specific user on a specific device. Think of it as a **telephone handset**—it's the physical connection through which a user participates in conversations. Each `ClientSession` links a low-level WebSocket object to a specific user identity (`userId`) and contains metadata about the connection itself.

Field Name	Type	Description & Constraints
`connectionId`	`string`	Unique identifier for this specific WebSocket connection. Generated when the connection is established (e.g., using a UUID). Used to reference this specific session in message routing and disconnection handling.
`socket`	`WebSocket`	The actual WebSocket object provided by your WebSocket library (e.g., `ws` in Node.js). This is the raw communication channel for sending and receiving data. The `readyState` property indicates if the connection is open, closing, or closed.
`userId`	`string`	The username of the authenticated user associated with this connection. References the `User.username` field. A single user may have multiple `ClientSession` objects if connected from multiple devices.
`ip`	`string`	The IP address of the client connection. Useful for logging, rate limiting, and security monitoring. Captured during the initial HTTP upgrade request.
`userAgent`	`string`	The User-Agent header from the client's initial HTTP request. Helps identify client types (browser, mobile app, bot) for debugging and analytics.

> **Design Insight:** Storing the raw `WebSocket` object directly in the `ClientSession` is a pragmatic choice for this learning-focused implementation. In production systems, you might wrap the socket with additional abstractions to handle reconnections, message batching, and protocol upgrades, but for clarity and simplicity, we keep the direct reference.

The relationship between these core types is visualized in the following diagram, which shows how users connect via sessions, join rooms, and send messages:



Wire Format and Event Protocol

If the core types are the DNA, the wire format is the **language spoken between clients and servers**. All communication over WebSocket connections follows a consistent JSON-based protocol where clients and servers exchange structured "events" with typed payloads. This protocol defines the vocabulary and grammar of your chat application—what can be said, how it's structured, and what responses are expected.

> **Mental Model:** Think of the WebSocket connection as a two-way radio channel. Clients and servers don't just send raw text back and forth—they send structured "radio messages" with a standard format: "This is Unit A to Base, message type: STATUS_REPORT, payload: {battery: 85%, location: grid B7}". The message type tells the receiver how to interpret the payload, and both sides agree on the meaning of each message type beforehand.

WebSocketMessage Envelope

Every message sent over the WebSocket connection, regardless of direction (client→server or server→client), is wrapped in a standard envelope structure called `WebSocketMessage`. This envelope provides metadata about the message content and enables extensibility for future message types.

Field	Type	Required	Description
`type`	`string`	Yes	Identifies the kind of event or action this message represents. Must be one of the predefined `EVENT_*` constants (e.g., `EVENT_CHAT_MESSAGE`, `EVENT_USER_JOINED`). This field tells the receiver which handler to invoke and how to parse the payload.
`payload`	`any` (JSON-serializable)	Yes	The data content specific to this message type. Structure varies by message type—see the Event Catalog below. Must be valid JSON (objects, arrays, strings, numbers, booleans, null).
`timestamp`	`string` (ISO 8601)	No (optional)	When the sender created this message. For server-originated messages, this is always populated. For client messages, the server will typically ignore client-provided timestamps and assign its own upon receipt to maintain a single source of truth.

Example Wire Format:

```json
{
  "type": "chat_message",
  "payload": {
    "sender": "alice",
    "content": "Hello everyone!",
    "roomId": "general"
  },
  "timestamp": "2024-01-15T10:30:45.123Z"
}
```

> **Design Insight:** Using a consistent envelope format (`type` + `payload`) rather than free-form JSON messages makes the protocol self-describing and extensible. New message types can be added without breaking existing clients (as long as they ignore unknown types), and the type field enables efficient routing to appropriate handler functions.

Event Catalog

The following table catalogues the standard event types that form the complete vocabulary of the chat protocol. These constants should be defined in your codebase and used consistently by both client and server implementations.

Event Constant	Value	Direction	Purpose & Payload Structure
`EVENT_CHAT_MESSAGE`	`"chat_message"`	Bidirectional	**Client→Server:** Sends a new chat message to a room. Payload: `{sender: string, content: string, roomId: string}` . **Server→Client:** Broadcasts a received message to all room members (excluding sender). Payload: A complete `ChatMessage` object with server-assigned `timestamp` .
`EVENT_USER_JOINED`	`"user_joined"`	Server→Client	Notifies existing room members that a new user has entered. Payload: `{username: string, roomId: string, timestamp: string}` . Typically sent after successful room join.
`EVENT_USER_LEFT`	`"user_left"`	Server→Client	Notifies room members that a user has left the room (either voluntarily or via disconnect). Payload: `{username: string, roomId: string, timestamp: string}` .
`EVENT_USER_TYPING`	`"user_typing"`	Bidirectional	**Client→Server:** Indicates the user has started typing in a room. Payload: `{username: string, roomId: string, isTyping: boolean}` (where `isTyping` is `true` for start, `false` for stop). **Server→Client:** Broadcasts typing status to other room members. Payload same as client→server.
`EVENT_JOIN_ROOM`	`"join_room"`	Client→Server	Requests to join a specific chat room. Payload: `{roomId: string, username: string}` . Server validates authentication and room existence before granting access.
`EVENT_JOIN_ROOM_SUCCESS`	`"join_room_success"`	Server→Client	Response confirming successful room entry. Payload: `{roomId: string, roomName: string, members: string[], history: ChatMessage[]}` . Includes current member list and recent message history.
`EVENT_JOIN_ROOM_ERROR`	`"join_room_error"`	Server→Client	Response indicating room join failed. Payload: `{roomId: string, error: string}` with human-readable error message.
`EVENT_ROOM_LIST`	`"room_list"`	Server→Client	Response to room listing request. Payload: `{rooms: Array<{id: string, name: string, memberCount: number}>}` .
`EVENT_PRESENCE_UPDATE`	`"presence_update"`	Server→Client	Broadcasts changes in user online/offline status across the system (not room-specific). Payload: `{username: string, isOnline: boolean, lastSeen: string}` .
`EVENT_ERROR`	`"error"`	Server→Client	Generic error response for malformed messages, authentication failures, or server errors. Payload: `{code: string, message: string, originalEventType: string}` .
`EVENT_HEARTBEAT`	`"heartbeat"`	Bidirectional	Periodic ping/pong messages to keep connection alive and detect disconnections. Payload: `{sequence: number}` (monotonically increasing number).

Architecture Decision: Structured Events vs. Ad-Hoc Messages

> **Decision: Use Typed JSON Event Protocol**
>
> - **Context:** We need a communication protocol between client and server over WebSocket that supports multiple message types (chat, presence, typing indicators, etc.) with varying payload structures. The protocol must be easy to implement, debug, and extend.
> - **Options Considered:**
>   1. **Free-form JSON with implicit type:** Send JSON objects with varying structures and infer meaning from field presence (e.g., `{text: "hello"}` is a chat message, `{typing: true}` is a typing indicator).
>   2. **String prefix protocol:** Send messages as strings with a type prefix and delimiter (e.g., `"CHAT|alice|Hello"` or `"TYPING|alice|true"`).
>   3. **Structured JSON with explicit type field:** Send JSON objects with a `type` field identifying the message type and a `payload` field containing type-specific data.
> - **Decision:** Option 3—structured JSON with explicit type field.
> - **Rationale:**
>   - **Self-documenting:** The `type` field makes messages immediately understandable during debugging.
>   - **Extensible:** New message types can be added without breaking existing parsing logic.
>   - **Type-safe:** On TypeScript clients, we can define discriminated unions based on the `type` field for compile-time validation.
>   - **Consistent with industry patterns:** Similar to Redux actions, Socket.io events, and other event-driven architectures developers may encounter.
> - **Consequences:**
>   - Slightly larger payload size due to the additional `type` field and nested `payload` structure.
>   - Requires proper validation of the `type` field against allowed values.
>   - Encourages a clean separation between message routing logic and message handling logic.

Option	Pros	Cons	Why Not Chosen
Free-form JSON	Minimal overhead, simple to create	Hard to extend, difficult to debug, ambiguous parsing	Too fragile—adding new message types requires modifying parsing logic
String prefix	Very compact, fast parsing	Not self-describing, hard to nest complex data	Poor developer experience, error-prone string manipulation
Structured JSON	**Self-documenting, extensible, type-safe**	**Slightly larger payloads**	**CHOSEN: Best balance of clarity and practicality**

Common Pitfalls: Protocol Design Mistakes

⚠️ Pitfall: Inconsistent Payload Structures

- **Description:** Using different field names or structures for the same event type in different parts of the codebase (e.g., sending `user` in some places but `username` in others for `EVENT_USER_JOINED`).
- **Why it's wrong:** Clients break because they expect consistent payloads. Debugging becomes difficult as you need to trace where each variant originates.
- **Fix:** Define payload interfaces/types for each event and reuse them everywhere. Consider creating factory functions that generate properly structured events.

⚠️ Pitfall: Missing Timestamps on Server-Originated Events

- **Description:** Forgetting to include the `timestamp` field in messages sent from server to client.
- **Why it's wrong:** Clients cannot properly order messages from different sources or display "time ago" indicators. Message history becomes ambiguous.
- **Fix:** Always add a `timestamp` field with `new Date().toISOString()` for server-originated messages. Create a helper function `createServerMessage(type, payload)` that automatically adds the timestamp.

⚠️ Pitfall: Not Validating Client-Provided Payloads

- **Description:** Trusting that clients will always send valid, well-formed payloads according to the expected schema.
- **Why it's wrong:** Malicious clients or buggy implementations can send malformed data that crashes the server or causes undefined behavior.
- **Fix:** Validate every incoming message against a schema (using libraries like `joi`, `zod`, or manual checks). Check required fields, data types, string lengths, and value ranges before processing.

⚠️ Pitfall: Ignoring Unknown Message Types

- **Description:** When receiving a message with an unrecognized `type` field, throwing an error or crashing.
- **Why it's wrong:** Prevents forward compatibility—if you deploy a server with new message types, older clients will crash when they receive messages they don't understand.
- **Fix:** Log a warning for unknown message types but don't crash. Silently ignore or send a generic "unsupported message type" error response.

Implementation Guidance

> **Target Language:** JavaScript/Node.js (with TypeScript types for clarity where helpful)

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
Message Serialization	Native `JSON.parse()` and `JSON.stringify()`	Schema validation with `zod` or `joi` before parsing
Type Safety	Manual type checking with `typeof` and `Array.isArray()`	TypeScript with strict mode and discriminated unions
Date Handling	JavaScript `Date` objects and `toISOString()`	`date-fns` or `luxon` for robust date manipulation
Unique IDs	`crypto.randomUUID()` (Node.js 15+) or `uuid` package	ULID or NanoID for time-ordered, URL-safe IDs

B. Recommended File/Module Structure

Place data model definitions and protocol constants in a dedicated module for easy import throughout your codebase:

```
real-time-chat/
├── package.json
├── src/
│   ├── server/
│   │   ├── index.js              # Server entry point
│   │   ├── connectionManager.js  # Component Design: WebSocket Server
│   │   ├── roomManager.js        # Component Design: Message Broker & Room Manager
│   │   ├── authService.js        # Component Design: Persistence & Authentication
│   │   └── models/               # DATA MODEL FILES GO HERE
│   │       ├── types.js          # Core type definitions (ChatMessage, User, etc.)
│   │       ├── protocol.js       # Event constants and wire format utilities
│   │       └── validation.js     # Schema validation for incoming messages
│   ├── client/
│   │   └── public/               # HTML/JS client files
│   └── shared/                   # Code shared between client and server
│       └── protocol.js           # Shared event constants (if using module bundler)
└── tests/
    └── unit/
        └── models/               # Tests for data model validation
```

C. Infrastructure Starter Code

Create a complete, reusable module for message validation and serialization. This code handles the boilerplate of parsing JSON, validating structure, and ensuring type safety:

File: `src/server/models/validation.js`

JAVASCRIPT

```javascript
/**
 * Validation utilities for WebSocket messages and data models
 */

const MAX_MESSAGE_LENGTH = 1000;

/**
 * Validates a raw WebSocket message string as a valid WebSocketMessage envelope
 * @param {string} rawMessage - Raw string received from WebSocket
 * @returns {{type: string, payload: any, timestamp?: string} | null}
 * @throws {Error} If message is not valid JSON or lacks required fields
 */
function parseAndValidateWebSocketMessage(rawMessage) {
  try {
    const parsed = JSON.parse(rawMessage);

    // Validate required fields
    if (typeof parsed.type !== 'string' || parsed.type.trim() === '') {
      throw new Error('Message must have a non-empty string "type" field');
    }

    if (parsed.payload === undefined) {
      throw new Error('Message must have a "payload" field');
    }

    // Optional timestamp validation if present
    if (parsed.timestamp !== undefined) {
      if (typeof parsed.timestamp !== 'string') {
        throw new Error('Timestamp must be a string if provided');
      }
      // Validate ISO format (basic check)
      if (!/\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}/.test(parsed.timestamp)) {
        throw new Error('Timestamp must be in ISO 8601 format');
      }
    }

    return {
      type: parsed.type.trim(),
      payload: parsed.payload,
      timestamp: parsed.timestamp
    };
```

```
  } catch (error) {
    // Enhance error message for JSON parsing errors
    if (error instanceof SyntaxError) {
      throw new Error(`Invalid JSON: ${error.message}`);
    }
    throw error;
  }
}

/**
 * Validates a ChatMessage payload structure
 * @param {any} payload - The payload to validate
 * @returns {{sender: string, content: string, roomId: string, timestamp: string}}
 * @throws {Error} If payload does not match ChatMessage structure
 */
function validateChatMessagePayload(payload) {
  if (typeof payload !== 'object' || payload === null) {
    throw new Error('Chat message payload must be an object');
  }

  const { sender, content, roomId, timestamp } = payload;

  if (typeof sender !== 'string' || sender.trim() === '') {
    throw new Error('Chat message must have a non-empty sender field');
  }

  if (typeof content !== 'string') {
    throw new Error('Chat message must have a string content field');
  }

  if (content.length > MAX_MESSAGE_LENGTH) {
    throw new Error(`Message content exceeds maximum length of ${MAX_MESSAGE_LENGTH} characters`);
  }

  if (typeof roomId !== 'string' || roomId.trim() === '') {
    throw new Error('Chat message must have a non-empty roomId field');
  }

  // Server should assign timestamp, but validate if provided
  if (timestamp !== undefined && typeof timestamp !== 'string') {
    throw new Error('Chat message timestamp must be a string if provided');
```

```
  }


  return {
    sender: sender.trim(),
    content: content.trim(),
    roomId: roomId.trim(),
    timestamp: timestamp || new Date().toISOString()
  };
}

/**
 * Creates a properly formatted WebSocketMessage for server-originated events
 * @param {string} type - Event type (use protocol constants)
 * @param {any} payload - Event-specific payload
 * @returns {string} JSON string ready to send over WebSocket
 */
function createServerMessage(type, payload) {
  const message = {
    type,
    payload,
    timestamp: new Date().toISOString()
  };
  return JSON.stringify(message);
}

module.exports = {
  parseAndValidateWebSocketMessage,
  validateChatMessagePayload,
  createServerMessage,
  MAX_MESSAGE_LENGTH
};
```

D. Core Logic Skeleton Code

Here are the skeleton implementations for the key data model handling functions that you'll need to implement:

File: `src/server/models/types.js`

JAVASCRIPT

```javascript
/**
 * Core type definitions and factory functions
 */

// Event constants - must match exactly with NAMING CONVENTIONS
const EVENT_CHAT_MESSAGE = 'chat_message';
const EVENT_USER_JOINED = 'user_joined';
const EVENT_USER_TYPING = 'user_typing';
// Add other event constants from the Event Catalog here...

/**
 * Creates a new ChatMessage object with server-assigned timestamp
 * @param {string} sender - Username of the sender
 * @param {string} content - Message content
 * @param {string} roomId - Room identifier
 * @returns {ChatMessage} A complete ChatMessage object
 */
function createChatMessage(sender, content, roomId) {
  // TODO 1: Validate that sender is a non-empty string
  // TODO 2: Validate that content is a string and doesn't exceed MAX_MESSAGE_LENGTH
  // TODO 3: Validate that roomId is a non-empty string
  // TODO 4: Generate a timestamp using new Date().toISOString()
  // TODO 5: Return an object with fields: sender, content, timestamp, roomId
  // TODO 6: Ensure all string fields are trimmed of surrounding whitespace
}

/**
 * Creates a new Room object with initial empty member set
 * @param {string} name - Human-readable room name
 * @returns {Room} A new Room object with generated id and creation timestamp
 */
function createRoom(name) {
  // TODO 1: Validate that name is a non-empty string
  // TODO 2: Generate a unique room ID (consider using crypto.randomUUID() or a URL-safe slug)
  // TODO 3: Create a new empty Set for memberIds
  // TODO 4: Record current timestamp as createdAt (Date object)
  // TODO 5: Return an object with fields: id, name, createdAt, memberIds
  // TODO 6: Sanitize room name to prevent injection attacks (remove special chars)
}

/**
 * Creates a ClientSession object linking a WebSocket to a user
```

```
 * @param {WebSocket} socket - The WebSocket connection object
 * @param {string} userId - Authenticated username
 * @param {string} ip - Client IP address
 * @param {string} userAgent - Client User-Agent header
 * @returns {ClientSession} A new ClientSession object
 */
function createClientSession(socket, userId, ip, userAgent) {
  // TODO 1: Validate that socket is a valid WebSocket object
  // TODO 2: Validate that userId is a non-empty string
  // TODO 3: Generate a unique connectionId (consider using crypto.randomUUID())
  // TODO 4: Return an object with fields: connectionId, socket, userId, ip, userAgent
  // TODO 5: Store the IP and UserAgent as provided (no need to validate format)
}

module.exports = {
  EVENT_CHAT_MESSAGE,
  EVENT_USER_JOINED,
  EVENT_USER_TYPING,
  createChatMessage,
  createRoom,
  createClientSession
};
```

E. Language-Specific Hints

1. **Use `Map` for Connection Tracking:** When storing `ClientSession` objects by `connectionId`, use a `Map` instead of a plain object for better performance with frequent additions/deletions and built-in iteration methods.
2. **Validate Early and Often:** Always validate incoming WebSocket messages immediately after parsing. Don't trust client data. Use the validation utilities provided above.
3. **ISO Timestamps for Consistency:** Always use `new Date().toISOString()` for timestamps in wire format. This ensures UTC timezone and consistent format across all clients.
4. **Trim User Input:** Always call `.trim()` on string fields from clients (usernames, room names, message content) to remove accidental leading/trailing whitespace.
5. **Use Sets for Membership:** When tracking room members, use JavaScript's `Set` object for O(1) membership tests and automatic deduplication.
6. **Weak References for Large Data:** Consider using `WeakMap` or `WeakSet` if you need to associate metadata with WebSocket objects without preventing garbage collection, though for this learning project, a regular `Map` is fine.

F. Milestone Checkpoint

After implementing the data model and protocol:

1. **Start your server** and open the browser console on your client page.
2. **Manually test the WebSocket connection** by creating a raw WebSocket message in the console:

```javascript
// Assuming your WebSocket is stored in a variable named `ws`

const testMessage = {
  type: 'chat_message',
  payload: {
    sender: 'testuser',
    content: 'Hello world!',
    roomId: 'general'
  }
};

ws.send(JSON.stringify(testMessage));
```

3. **Expected behavior:**
   - Server should log the parsed message with type and payload
   - Server should validate the message structure
   - Server should reject malformed JSON with an error response
   - Server should reject messages missing required fields
4. **Signs something is wrong:**
   - **Server crashes on message:** Check your JSON parsing error handling
   - **Message ignored silently:** Verify you're calling your message handlers
   - **Validation not working:** Test with intentionally bad data to ensure validation catches it

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Messages appear with wrong timestamp"	Client is sending timestamps and server is using them instead of assigning its own	Check server logs to see if incoming messages have timestamp fields	Always overwrite timestamp with server time in `createChatMessage`
"Duplicate users in room member list"	Using array instead of Set for `memberIds`, or not checking if user already exists before adding	Log room members before/after join operations	Use `Set` for `memberIds` and check `Set.has()` before adding
"Server crashes when receiving malformed JSON"	No try-catch around `JSON.parse()`	Send a non-JSON string like "invalid" to the WebSocket	Wrap `JSON.parse()` in try-catch and send error response
"Event types are case-sensitive mismatches"	Using different casing for constants in client vs server (e.g., 'chat_message' vs 'CHAT_MESSAGE')	Compare the `type` field values in server logs with your constant definitions	Use exactly the same string constants in both client and server code
"Very long messages crash the server"	Not validating message length before processing	Send a 10,000 character message	Add length check in `validateChatMessagePayload` against `MAX_MESSAGE_LENGTH`

Component Design: WebSocket Server & Connection Manager

> **Milestone(s):** Milestone 1 (WebSocket Server Setup). This component forms the foundational communication layer of the entire system, responsible for establishing and maintaining persistent connections with all clients.

Responsibility and Scope

The **WebSocket Server & Connection Manager** is the gateway and traffic controller for all real-time communication in the chat application. It sits at the network boundary and has three core responsibilities:

1. **Protocol Handling**: Manages the low-level WebSocket protocol, including accepting HTTP upgrade requests, establishing persistent connections, and handling the raw binary/text frames defined by RFC 6455.

2. **Connection State Management**: Maintains a real-time inventory of all active client connections, tracking their lifecycle from establishment through normal operation to termination (whether graceful or abrupt).
3. **Message Routing Foundation**: Acts as the initial entry point for all incoming real-time traffic, performing basic validation and routing messages to the appropriate higher-level business logic components (like the Room Manager).

This component **owns** the following:

- The raw WebSocket connection objects for each client
- The mapping between connection identifiers and their underlying sockets
- Basic connection metadata (IP address, user agent, connection timestamp)
- The heartbeat/ping-pong mechanism for detecting dead connections

This component **does NOT own**:

- User authentication state (delegated to Auth Service)
- Room membership information (delegated to Room Manager)
- Message persistence (delegated to Persistence Service)
- Business logic for chat operations (delegated to appropriate handlers)

The scope is deliberately narrow: think of this component as the telephone company's switching hardware—it ensures calls can be connected and stay connected, but doesn't understand the content of the conversations or who should be talking to whom.

Mental Model: The Telephone Switchboard Operator

Imagine a bustling hotel in the 1950s. Guests (clients) arrive and want to make phone calls to other guests' rooms. The **switchboard operator** (our WebSocket Server) has a physical panel with:

- **Jacks (sockets)**: One for each guest room that wants phone service
- **Cords (connections)**: Physical wires connecting callers
- **Indicator lights (heartbeats)**: Showing which lines are still active

When a new guest checks in (client connects), the operator plugs a jack into their room's socket. When the guest picks up the phone and says "Connect me to Room 237" (client sends a `join` message), the operator doesn't just blindly connect wires—they check if Room 237 exists (room validation), verify the guest is allowed to call there (authentication), then physically connect the cords (routing).

The operator maintains a **switchboard directory** (connection map) showing which jack corresponds to which room. If a light goes out (connection drops), the operator immediately removes that jack from the board and notifies anyone who might be trying to call that room (cleanup and notification).

This mental model captures several key concepts:

- **Persistent connections** = physical jacks plugged in
- **Message routing** = operator connecting the right cords
- **Connection tracking** = the switchboard directory
- **Heartbeat detection** = indicator lights showing line activity
- **Graceful degradation** = operator can still work if some lights fail

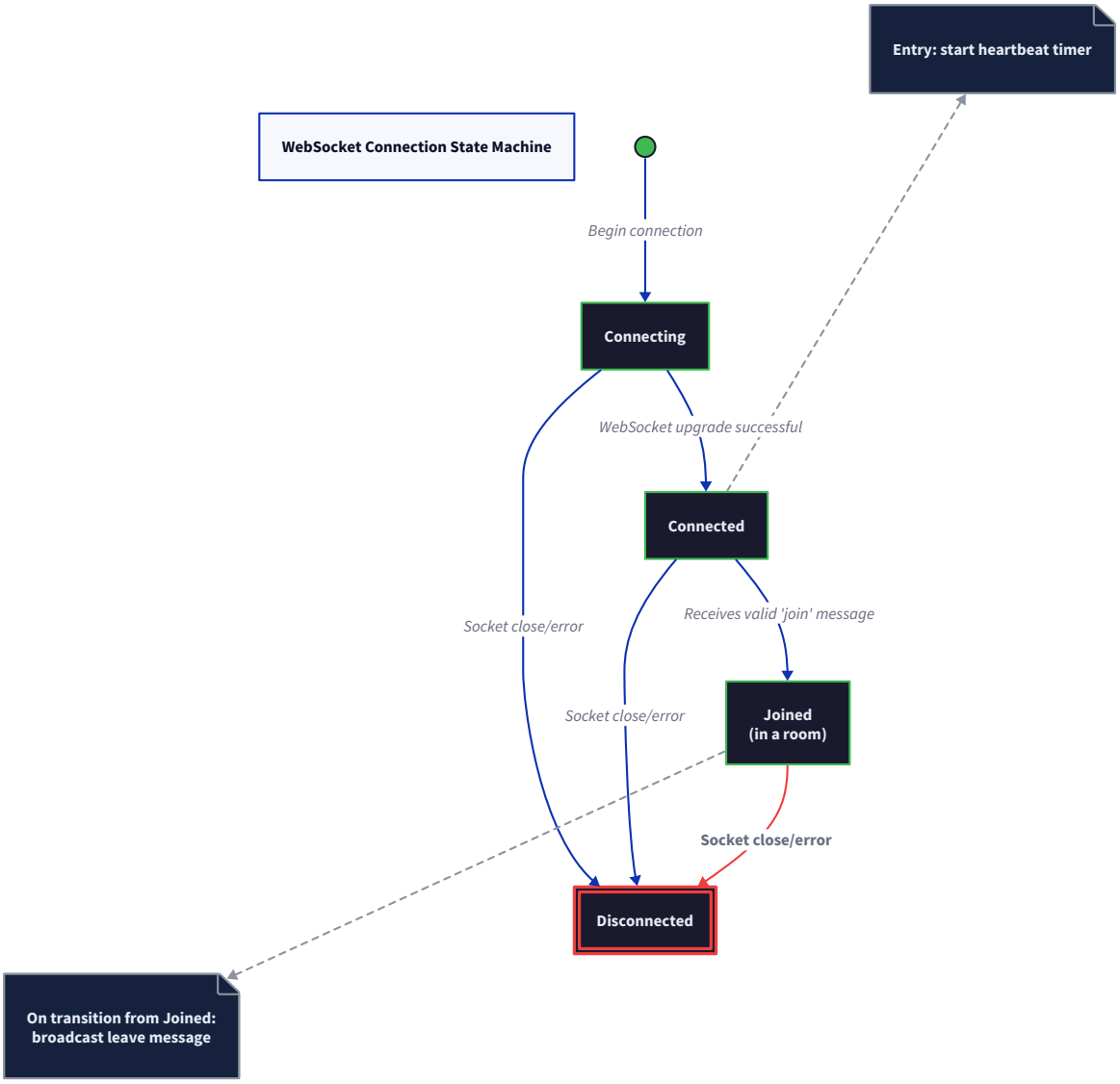
> **Design Insight**: The WebSocket Server is infrastructure, not business logic. Its job is to move bytes reliably, not understand what those bytes mean. This separation allows the business logic (rooms, messages, users) to evolve independently of the communication layer.

Interface and Connection Lifecycle

Every WebSocket connection follows a predictable lifecycle with distinct states and events. The server must handle each transition appropriately.

Connection State Machine

The following table describes the complete state machine for a client connection. Reference the state diagram:


WebSocket Connection State Machine
Entry: start heartbeat timer
Begin connection
Connecting
WebSocket upgrade successful
Connected
Socket close/error
Receives valid 'join' message
Socket close/error
Joined
(in a room)
Socket close/error
Disconnected
On transition from Joined:
broadcast leave message

Current State	Event Trigger	Next State	Actions Taken
None	HTTP Upgrade request received	`Connecting`	1. Validate HTTP headers 2. Check authentication token (if required) 3. Generate unique `connectionId` 4. Send 101 Switching Protocols response
`Connecting`	WebSocket handshake completes successfully	`Connected`	1. Create `ClientSession` object 2. Add to connection tracking map 3. Start heartbeat timer (ping/pong) 4. Log connection event 5. Send `connection_established` event to client
`Connected`	Client sends valid `join_room` message	`Joined`	1. Validate room exists/user can join 2. Register with Room Manager 3. Update session with `roomId` 4. Send `room_joined` confirmation with history
`Joined`	Client sends `leave_room` message	`Connected`	1. Unregister from Room Manager 2. Clear session `roomId` 3. Send `room_left` confirmation
`Connected` or `Joined`	Client sends `chat_message`	*Same state*	1. Parse and validate message 2. Forward to Room Manager for broadcasting 3. Acknowledge receipt to sender
`Connected` or `Joined`	Ping timeout (no pong received)	`Disconnected`	1. Mark connection as dead 2. Clean up from all tracking structures 3. Notify Room Manager of user departure
`Connected` or `Joined`	Socket `close` event received	`Disconnected`	1. Remove from connection map 2. Clear heartbeat timer 3. Notify Room Manager (if joined)
`Connected` or `Joined`	Socket `error` event	`Disconnected`	1. Log error details 2. Force socket closure 3. Perform same cleanup as `close` event
`Disconnected`	Cleanup completed	*None*	Memory released, no further actions

Core Interface Methods

The Connection Manager exposes the following public API to other components:

Method Name	Parameters	Returns	Description
`startServer`	`port: number`, `options: Object`	`Promise<void>`	Binds to specified port, starts listening for HTTP/WebSocket connections
`handleUpgrade`	`request: HTTPRequest`, `socket: NetSocket`, `head: Buffer`	`void`	Processes HTTP upgrade request, validates, establishes WebSocket connection
`handleMessage`	`socket: WebSocket`, `rawMessage: string	Buffer`	`void`
`sendToClient`	`connectionId: string`, `message: WebSocketMessage`	`boolean`	Sends formatted message to specific client if socket is open; returns success/failure
`broadcastToAll`	`message: WebSocketMessage`, `excludeConnectionId?: string`	`void`	Sends message to all connected clients (rarely used directly)
`getActiveConnections`	-	`Map<string, ClientSession>`	Returns copy of current connection map for monitoring/debugging
`closeConnection`	`connectionId: string`, `code: number`, `reason: string`	`boolean`	Gracefully closes specified connection with WebSocket close frame
`getConnectionStats`	-	`{ total: number, byState: Object }`	Returns statistics about current connections

Event Handlers (Internal Interface)

The component reacts to these WebSocket-native events:

Event	Handler Method	Typical Actions
`connection`	`onConnection(socket, request)`	Create `ClientSession`, start heartbeat, log connection
`message`	`onMessage(socket, data)`	Call `handleMessage()`, validate, route to business logic
`close`	`onClose(socket, code, reason)`	Clean up connection state, notify Room Manager, log disconnection
`error`	`onError(socket, error)`	Log error, force closure, clean up (treat as disconnect)
`pong`	`onPong(socket, data)`	Update last activity timestamp, mark connection as alive

ADR: Managing Active Connections

> **Decision: Use Map<string, ClientSession> for Connection Tracking**
>
> - **Context**: The server needs to maintain real-time awareness of all connected clients to route messages efficiently and perform cleanup. Each connection must be uniquely identifiable and quickly accessible.
> - **Options Considered**:
>   1. **Set of WebSocket objects**: Simple collection of raw socket objects
>   2. **Array of connection objects**: Sequential storage with manual ID management
>   3. **Map<string, ClientSession>**: Key-value store with connection ID as key
> - **Decision**: Use `Map<string, ClientSession>` where key is a unique `connectionId` and value is a `ClientSession` object containing the socket and metadata.
> - **Rationale**:
>   - **Direct Access**: `Map.get(connectionId)` is O(1) for targeted message sending
>   - **Built-in Iteration**: `Map.forEach()` efficiently broadcasts to all connections
>   - **Automatic Cleanup**: Deleting entries is explicit and clear (`Map.delete()`)
>   - **Metadata Association**: `ClientSession` struct can hold user ID, IP, room ID, etc., alongside the raw socket
>   - **Memory Safety**: Unlike Sets with object references, string keys prevent memory leaks from object resurrection
> - **Consequences**:
>   - Adds slight memory overhead for the `ClientSession` wrapper
>   - Requires generating and managing unique connection IDs
>   - Provides excellent foundation for future features like direct messaging and connection monitoring

The following comparison table illustrates the trade-offs:

Option	Pros	Cons	Why Not Chosen
Set of WebSocket objects	Simple, no ID management, native WebSocket API	No metadata storage, O(n) lookup for specific connections, harder to debug	Insufficient for routing needs; can't associate user ID with socket
Array of connection objects	Maintains order, simple iteration	O(n) lookup, manual ID management, inefficient removals	Performance degrades with hundreds of connections
Map<string, ClientSession>	O(1) lookup, clean metadata association, easy iteration	Requires ID generation, wrapper object overhead	**CHOSEN**: Best balance of performance, functionality, and clarity

Common Pitfalls: Memory Leaks and Silent Failures

Building a WebSocket server involves several subtle traps that can cause gradual degradation or sudden failure. Here are the most common pitfalls and how to avoid them:

⚠️ Pitfall 1: Zombie Connections - Not Cleaning Up on Disconnect

The Mistake: Only removing connections on explicit `close` events, ignoring error conditions, timeouts, or network interruptions that don't trigger proper close frames. **Why It's Wrong**: Dead connections accumulate in the tracking map, causing memory leaks. The server continues trying to send messages to dead sockets, wasting CPU and potentially crashing when buffers fill. **How to Fix**: Implement a **defense-in-depth cleanup strategy**:

1. Always remove from tracking in BOTH `close` and `error` handlers
2. Implement heartbeat (ping/pong) with timeout detection
3. Add periodic sanity checks that iterate through connections and verify socket state
4. Use weak references or finalizers if available (though JavaScript lacks these)

⚠️ Pitfall 2: The Silent Swallow - Missing Error Handlers

The Mistake: Not attaching error handlers to WebSocket objects, assuming connections will always close gracefully.

```javascript
// BAD: Socket errors become unhandled exceptions
socket.on('message', (data) => { /* process */ });

// GOOD: Explicit error handling
socket.on('error', (err) => {
  console.error(`Socket error for ${connectionId}:`, err);
  this.closeConnection(connectionId, 1011, 'Internal error');
});
```

Why It's Wrong: Unhandled socket errors can crash the entire Node.js process. Even if they don't crash, failures become invisible, making debugging nearly impossible. **How to Fix**: Attach error handlers to EVERY socket, EVERY EventEmitter, and wrap ALL asynchronous operations in try/catch. Log errors with context (connectionId, user ID) for debugging.

⚠️ Pitfall 3: The Forgotten Heartbeat - No Liveness Detection

The Mistake: Assuming connections remain alive because the TCP socket is open, ignoring "half-open" connections where the client disappeared without closing. **Why It's Wrong**: Network issues, client crashes, or NAT timeouts can leave sockets in a "zombie" state—technically open but unable to send/receive data. The server thinks users are online when they're not. **How to Fix**: Implement **RFC 6455 ping/pong frames**:

1. Periodically send ping frames to each connection (every 30-60 seconds)
2. Set a timer expecting a pong response within 10-15 seconds
3. Close connections that miss too many pongs (typically 2-3 consecutive failures)
4. Update `lastActive` timestamp on any received message (including pong)

⚠️ Pitfall 4: Ready State Race Condition - Sending to Closing Sockets

The Mistake: Not checking `socket.readyState` before sending, causing "Cannot send after socket closed" errors. **Why It's Wrong**: Between checking if a connection exists and actually sending, the socket could close (especially under load). This throws unhandled exceptions. **How to Fix**: Always verify ready state AND catch send errors:

```javascript
sendToClient(connectionId, message) {
  const session = this.connections.get(connectionId);
  if (!session) return false;

  // Check state before sending
  if (session.socket.readyState !== WebSocket.OPEN) {
    this.cleanupConnection(connectionId);
    return false;
  }

  try {
    session.socket.send(JSON.stringify(message));
    return true;
  } catch (err) {
    // Socket closed between check and send
    this.cleanupConnection(connectionId);
    return false;
  }
}
```

⚠️ Pitfall 5: Unbounded Message Queue - No Backpressure Handling

The Mistake: Accepting and queuing unlimited messages from clients without considering server capacity. **Why It's Wrong**: A malicious or buggy client could flood the server with messages, consuming memory and CPU, eventually causing out-of-memory crashes. **How to Fix**: Implement **per-connection backpressure**:

1. Track number of unprocessed messages per connection
2. Pause socket reception when backlog exceeds threshold (using `socket.pause()`)
3. Resume when backlog clears
4. Implement maximum message size validation (enforce `MAX_MESSAGE_LENGTH`)
5. Close connections that consistently exceed limits

⚠️ Pitfall 6: The Shared Mutability Trap - Concurrent Modification

The Mistake: Iterating over the connections Map while other code modifies it (adding/removing connections). **Why It's Wrong**: In Node.js's single-threaded event loop, this seems safe, but asynchronous callbacks can interleave, causing modification during iteration errors or skipped connections. **How to Fix**: Use defensive copying or proper locking:

```javascript
// SAFE: Create copy before iterating
broadcastToAll(message, excludeConnectionId) {
  const connectionsCopy = new Map(this.connections);
  connectionsCopy.forEach((session, connectionId) => {
    if (connectionId !== excludeConnectionId) {
      this.sendToClient(connectionId, message);
    }
  });
}
```

Implementation Guidance (Layer 2)

This section provides concrete implementation guidance for building the WebSocket Server & Connection Manager in JavaScript/Node.js, focusing on the `ws` library (the most popular WebSocket implementation for Node.js).

A. Technology Recommendations Table

Component	Simple Option (Learning)	Production-Ready Option
WebSocket Library	`ws` (minimal, RFC-compliant)	`ws` with extensions (`permessage-deflate` for compression)
HTTP Server	Node.js built-in `http` or `https`	Express.js + `express-ws` for API co-existence
Connection Tracking	In-memory `Map<string, ClientSession>`	Redis for distributed tracking (multi-server)
Heartbeat Mechanism	Manual ping/pong with `setInterval`	`ws` built-in ping/pong with configurable interval
Logging	`console.log` with timestamps	Structured logging (Winston, Pino) with connection context

B. Recommended File/Module Structure

```
real-time-chat/
├── package.json
├── tsconfig.json (if using TypeScript)
├── src/
│   ├── server.ts (or server.js)          # Main entry point
│   ├── core/
│   │   ├── ConnectionManager.ts          # THIS COMPONENT (primary file)
│   │   ├── types.ts                      # Shared type definitions
│   │   └── constants.ts                  `EVENT_*`, `MAX_MESSAGE_LENGTH`, etc.
│   ├── services/
│   │   ├── RoomManager.ts                # Milestone 2-3 component
│   │   └── AuthService.ts                # Milestone 4 component
│   ├── persistence/
│   │   └── MessageStore.ts               # Milestone 4 persistence
│   └── client/
│       └── index.html                    # Basic web client
└── tests/
    └── ConnectionManager.test.ts
```

C. Infrastructure Starter Code

Complete WebSocket Server Setup with HTTP Upgrade Handling

JAVASCRIPT

```javascript
// File: src/server.js

const http = require('http');

const WebSocket = require('ws');

const { ConnectionManager } = require('./core/ConnectionManager');

const { parseAndValidateWebSocketMessage } = require('./core/messageUtils');


// Create HTTP server (could be Express in more advanced setups)

const server = http.createServer((req, res) => {

  // Handle regular HTTP requests (health checks, static files, API)

  if (req.url === '/health') {

    res.writeHead(200, { 'Content-Type': 'application/json' });

    res.end(JSON.stringify({ status: 'ok', connections: connectionManager.getConnectionStats() }));

    return;

  }



  // Default 404 for other HTTP requests

  res.writeHead(404);

  res.end();

});


// Create WebSocket server attached to HTTP server

const wss = new WebSocket.Server({

  server,

  clientTracking: false, // We'll handle tracking ourselves

  maxPayload: 1024 * 1024, // 1MB max message size

});


// Initialize connection manager

const connectionManager = new ConnectionManager();


// Handle WebSocket connections

wss.on('connection', (socket, request) => {

  // Extract connection info from request

  const ip = request.headers['x-forwarded-for'] || request.socket.remoteAddress;

  const userAgent = request.headers['user-agent'] || 'unknown';



  // Generate unique connection ID

  const connectionId = generateConnectionId();



  // Create client session (user ID will be added after authentication in Milestone 4)

  const session = {

    connectionId,
```

```
    socket,
    userId: null, // Will be set after auth
    ip,
    userAgent,
    joinedAt: new Date().toISOString(),
    lastActivity: Date.now(),
    roomId: null, // Current room, if any
  };

  // Add to connection tracking
  connectionManager.addConnection(connectionId, session);

  console.log(`[${new Date().toISOString()}] Connection established: ${connectionId} from ${ip}`);

  // Send welcome message
  socket.send(JSON.stringify({
    type: 'connection_established',
    payload: { connectionId, serverTime: new Date().toISOString() },
  }));

  // Set up message handler
  socket.on('message', (rawData) => {
    try {
      // Update activity timestamp
      session.lastActivity = Date.now();

      // Parse and validate message
      const message = parseAndValidateWebSocketMessage(rawData.toString());

      // Route to appropriate handler
      connectionManager.handleMessage(connectionId, message);
    } catch (error) {
      console.error(`Error processing message from ${connectionId}:`, error);

      // Send error back to client
      socket.send(JSON.stringify({
        type: 'error',
        payload: { message: 'Invalid message format', details: error.message },
      }));
    }
```

```
  });

  // Set up close handler
  socket.on('close', (code, reason) => {
    console.log(`[${new Date().toISOString()}] Connection closed: ${connectionId}, code: ${code}, reason: ${reason}`);
    connectionManager.removeConnection(connectionId);
  });

  // Set up error handler
  socket.on('error', (error) => {
    console.error(`[${new Date().toISOString()}] Socket error for ${connectionId}:`, error);
    connectionManager.removeConnection(connectionId);
    socket.terminate(); // Force close
  });

  // Set up pong handler for heartbeat
  socket.on('pong', () => {
    session.lastActivity = Date.now();
  });
});

// Start heartbeat mechanism
setInterval(() => {
  connectionManager.checkHeartbeats();
}, 30000); // Check every 30 seconds

// Start server
const PORT = process.env.PORT || 8080;
server.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
  console.log(`WebSocket endpoint: ws://localhost:${PORT}`);
});

// Helper function to generate connection IDs
function generateConnectionId() {
  return `conn_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
}
```

Complete Message Validation Utilities

JAVASCRIPT

```javascript
// File: src/core/messageUtils.js

const constants = require('./constants');


/**

 * Parses and validates raw WebSocket message string

 * @param {string} rawMessage - Raw message from WebSocket

 * @returns {WebSocketMessage} - Parsed and validated message object

 * @throws {Error} - If message is invalid JSON or missing required fields

 */

function parseAndValidateWebSocketMessage(rawMessage) {

  // Check message length

  if (rawMessage.length > constants.MAX_MESSAGE_LENGTH) {

    throw new Error(`Message exceeds maximum length of ${constants.MAX_MESSAGE_LENGTH} characters`);

  }


  let parsed;

  try {

    parsed = JSON.parse(rawMessage);

  } catch (error) {

    throw new Error('Invalid JSON format');

  }


  // Validate required fields

  if (typeof parsed.type !== 'string' || parsed.type.trim() === '') {

    throw new Error('Message must have a non-empty "type" field');

  }


  if (parsed.payload === undefined) {

    throw new Error('Message must have a "payload" field');

  }


  // Add timestamp if not present

  if (!parsed.timestamp) {

    parsed.timestamp = new Date().toISOString();

  }


  return {

    type: parsed.type,

    payload: parsed.payload,

    timestamp: parsed.timestamp,

  };
```

```
}

/**
 * Validates ChatMessage payload structure
 * @param {any} payload - The payload to validate
 * @returns {boolean} - True if valid
 * @throws {Error} - If validation fails
 */
function validateChatMessagePayload(payload) {
  if (typeof payload !== 'object' || payload === null) {
    throw new Error('Payload must be an object');
  }

  if (typeof payload.content !== 'string' || payload.content.trim() === '') {
    throw new Error('Message content must be a non-empty string');
  }

  if (payload.content.length > constants.MAX_MESSAGE_LENGTH) {
    throw new Error(`Message content exceeds ${constants.MAX_MESSAGE_LENGTH} characters`);
  }

  if (payload.roomId && typeof payload.roomId !== 'string') {
    throw new Error('roomId must be a string if provided');
  }

  if (payload.sender && typeof payload.sender !== 'string') {
    throw new Error('sender must be a string if provided');
  }

  return true;
}

/**
 * Creates properly formatted WebSocketMessage for server-originated events
 * @param {string} type - Event type (use constants.EVENT_*)
 * @param {any} payload - Message payload
 * @returns {WebSocketMessage} - Formatted message
 */
function createServerMessage(type, payload) {
  return {
    type,
```

```
    payload,
    timestamp: new Date().toISOString(),
  };
}

module.exports = {
  parseAndValidateWebSocketMessage,
  validateChatMessagePayload,
  createServerMessage,
};
```

D. Core Logic Skeleton Code

Connection Manager Class with TODOs

JAVASCRIPT

```javascript
// File: src/core/ConnectionManager.js

const WebSocket = require('ws');

const { createServerMessage } = require('./messageUtils');

const constants = require('./constants');


class ConnectionManager {

  constructor() {

    // ADR Decision: Using Map for connection tracking

    this.connections = new Map(); // connectionId -> ClientSession



    // In a real app, these would be injected dependencies

    this.roomManager = null; // Will be set after RoomManager is created

    this.authService = null; // Will be set in Milestone 4



    // Heartbeat configuration

    this.heartbeatInterval = 30000; // 30 seconds

    this.heartbeatTimeout = 10000; // 10 seconds timeout for pong

  }



  /**

   * Adds a new connection to tracking

   * @param {string} connectionId - Unique connection identifier

   * @param {ClientSession} session - Session object with socket and metadata

   */

  addConnection(connectionId, session) {

    // TODO 1: Validate parameters (connectionId must be string, session must have socket)

    // TODO 2: Check if connectionId already exists (log warning if it does)

    // TODO 3: Add to connections Map

    // TODO 4: Set up initial heartbeat timer

    // TODO 5: Log connection addition for debugging

  }



  /**

   * Removes a connection from tracking and cleans up resources

   * @param {string} connectionId - Connection to remove

   * @param {number} closeCode - WebSocket close code (optional)

   * @param {string} closeReason - Human-readable close reason (optional)

   */

  removeConnection(connectionId, closeCode, closeReason) {

    // TODO 1: Look up connection in Map

    // TODO 2: If connection was in a room, notify RoomManager to remove from room
```

```
  // TODO 3: If socket is still open, send close frame with provided code/reason
  // TODO 4: Clear any heartbeat timers for this connection
  // TODO 5: Remove from connections Map
  // TODO 6: Log removal with connectionId and reason
}

/**
 * Main entry point for handling incoming WebSocket messages
 * Reference flowchart: ./diagrams/flowchart-message-handling.svg
 * @param {string} connectionId - Which connection sent the message
 * @param {WebSocketMessage} message - Parsed and validated message
 */
handleMessage(connectionId, message) {
  // TODO 1: Look up connection in connections Map
  // TODO 2: Update lastActivity timestamp on the session
  // TODO 3: Route message based on type:
  //   - If type is 'join_room': call handleJoinRoom(connectionId, message.payload)
  //   - If type is 'leave_room': call handleLeaveRoom(connectionId, message.payload)
  //   - If type is 'chat_message': call handleChatMessage(connectionId, message.payload)
  //   - If type is 'typing_indicator': call handleTypingIndicator(connectionId, message.payload)
  //   - If type is 'ping': send 'pong' response immediately
  //   - Unknown type: send error response back to client
  // TODO 4: Catch any errors during processing and send appropriate error response
}

/**
 * Handles join room request from client
 * @param {string} connectionId - Connection wanting to join
 * @param {object} payload - Should contain roomId or roomName
 */
handleJoinRoom(connectionId, payload) {
  // TODO 1: Get connection from connections Map
  // TODO 2: Validate payload has roomId or roomName
  // TODO 3: If user is already in a room, call handleLeaveRoom first
  // TODO 4: Call roomManager.joinRoom(connectionId, payload.roomId, payload.roomName)
  // TODO 5: On success, update session.roomId
  // TODO 6: Send success response to client with room info and message history
  // TODO 7: On error, send error response to client
}
```

```
/**
 * Handles chat message from client
 * @param {string} connectionId - Connection that sent the message
 * @param {object} payload - Should contain content and optionally roomId
 */
handleChatMessage(connectionId, payload) {
  // TODO 1: Get connection from connections Map
  // TODO 2: Validate payload.content is non-empty string and within length limits
  // TODO 3: If payload.roomId is provided, use it; otherwise use session.roomId
  // TODO 4: Create ChatMessage object with sender from session.userId
  // TODO 5: Call roomManager.broadcastToRoom(roomId, chatMessage, excludeConnectionId)
  // TODO 6: Send acknowledgment back to sender
  // TODO 7: On error (e.g., not in a room), send error response
}

/**
 * Sends a message to a specific client
 * @param {string} connectionId - Target connection
 * @param {WebSocketMessage} message - Message to send
 * @returns {boolean} - True if sent successfully, false if failed
 */
sendToClient(connectionId, message) {
  // TODO 1: Look up connection in connections Map
  // TODO 2: If not found, return false
  // TODO 3: Check socket.readyState === WebSocket.OPEN
  // TODO 4: If not open, call removeConnection and return false
  // TODO 5: Try to send message with JSON.stringify
  // TODO 6: Catch any send errors, call removeConnection, return false
  // TODO 7: On success, return true
}

/**
 * Broadcasts message to all connected clients
 * @param {WebSocketMessage} message - Message to broadcast
 * @param {string} excludeConnectionId - Optional connection to exclude (e.g., sender)
 */
broadcastToAll(message, excludeConnectionId) {
  // TODO 1: Create a copy of connections Map to avoid modification during iteration
  // TODO 2: For each connection in the copy:
  //   - If connectionId !== excludeConnectionId, call sendToClient
```

```
    // TODO 3: Log broadcast statistics (how many sent, how many failed)
  }

  /**
   * Checks heartbeats for all connections and removes dead ones
   */
  checkHeartbeats() {
    const now = Date.now();
    const deadConnections = [];

    // TODO 1: Iterate through connections Map
    // TODO 2: For each connection, calculate time since lastActivity
    // TODO 3: If time > heartbeatInterval + heartbeatTimeout, mark as dead
    // TODO 4: Send ping to connections that haven't been pinged recently
    // TODO 5: For dead connections, add to deadConnections array

    // Clean up dead connections
    deadConnections.forEach(connectionId => {
      console.warn(`Connection ${connectionId} heartbeat failed, removing`);
      this.removeConnection(connectionId, 1001, 'Heartbeat timeout');
    });

    // TODO 6: Log heartbeat check results
  }

  /**
   * Gets connection statistics for monitoring
   * @returns {object} - Statistics about current connections
   */
  getConnectionStats() {
    // TODO 1: Initialize stats object with total count and by-state breakdown
    // TODO 2: Iterate through connections Map to collect:
    //   - Total connections
    //   - Connections with userId set (authenticated)
    //   - Connections with roomId set (in a room)
    //   - Connections grouped by userAgent (browser types)
    // TODO 3: Return statistics object
  }
}
```

```
module.exports = { ConnectionManager };
```

E. Language-Specific Hints (JavaScript/Node.js)

1. **WebSocket Library Choice**: Use `ws` (npm install ws). It's the most battle-tested WebSocket library for Node.js and implements the full RFC 6455 specification.
2. **Connection ID Generation**: Use a combination of timestamp and random string to avoid collisions:

```javascript
function generateId() {
  return `conn_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
}
```

3. **Handling Large Numbers of Connections**: Node.js can handle thousands of concurrent WebSocket connections, but you need to:
   - Increase the `ulimit` on Linux/Mac: `ulimit -n 10000`
   - Use `socket.setTimeout()` to detect stalled connections
   - Consider using the `cluster` module to utilize multiple CPU cores
4. **Memory Management**:
   - Regularly check `process.memoryUsage()` in production
   - Remove all references to sockets when connections close
   - Consider using `WeakMap` for auxiliary data if you don't need to iterate over it
5. **Error Handling Best Practices**:

```javascript
// Always attach error handlers
socket.on('error', (error) => {
  // Don't throw, log and clean up
  console.error('Socket error:', error);
  this.cleanupConnection(connectionId);
});

// Handle async errors in message processing
try {
  await someAsyncOperation();
} catch (error) {
  // Send error to client instead of crashing
  socket.send(JSON.stringify({ type: 'error', payload: { message: error.message } }));
}
```

6. **Testing WebSocket Servers**: Use `ws` client in tests:

```javascript
const WebSocket = require('ws');


test('server accepts connections', async () => {
  const ws = new WebSocket('ws://localhost:8080');


  await new Promise((resolve) => {
    ws.on('open', resolve);
  });


  expect(ws.readyState).toBe(WebSocket.OPEN);
  ws.close();
});
```

F. Milestone Checkpoint: WebSocket Server Setup

After implementing the Connection Manager:

Expected Behavior:

1. Start the server: `node src/server.js`
2. Open browser console at `http://localhost:8080` (you'll need a basic HTML file)
3. Connect via WebSocket: `new WebSocket('ws://localhost:8080')`
4. See connection established message in browser console
5. See connection log in server terminal

Verification Steps:

1. **Multiple Connections**: Open multiple browser tabs, verify each gets unique connection ID
2. **Heartbeat Detection**: Wait 45+ seconds without sending messages, verify connections stay alive (pings sent)
3. **Clean Disconnect**: Close a browser tab, verify server logs the disconnect and removes from tracking
4. **Error Handling**: Send malformed JSON message, verify server sends error response instead of crashing
5. **Memory Check**: Connect/disconnect 10+ times rapidly, verify connection count returns to zero

Signs of Problems:

- **"Too many open files" error**: Need to increase system ulimit
- **Memory keeps growing**: Connections aren't being cleaned up properly
- **Server crashes on message send**: Missing `readyState` check or error handler
- **"Cannot send after socket closed"**: Race condition between check and send

Debugging Commands:

```bash
# Check active connections
curl http://localhost:8080/health


# Monitor memory usage
node -e "setInterval(() => console.log(process.memoryUsage()), 5000)" &


# Test WebSocket connection manually (install wscat)
npm install -g wscat
wscat -c ws://localhost:8080
```

Component Design: Message Broker & Room Manager

> **Milestone(s):** Milestone 2 (Message Broadcasting), Milestone 3 (Chat Rooms)

This component is the heart of the chat application's business logic. While the WebSocket Server & Connection Manager handles the low-level plumbing of persistent connections, the Message Broker & Room Manager orchestrates the meaningful interactions within those connections: who can talk to whom, where messages should go, and who is currently active. It transforms raw WebSocket connections into a structured chat system with rooms, presence, and targeted message delivery.

Responsibility and Scope

The **Message Broker & Room Manager** (hereafter "Room Manager") is a stateful server-side component that sits atop the connection layer. Its core responsibilities are:

1. **Room Management:** Creating, tracking, and destroying logical chat rooms (`Room` objects). A room is a named channel where messages are broadcast to all members.
2. **Membership Management:** Maintaining, for each room, the set of active users (represented by their `connectionId`s). It adds users when they join and removes them when they leave or disconnect.
3. **Message Routing:** Receiving incoming `ChatMessage` events and delivering them to all current members of the relevant room, excluding the original sender (unless specified otherwise). This is the "broadcast" pattern.
4. **Presence Tracking:** Maintaining and broadcasting real-time status information about users. This includes:
   - **Online/Offline:** Tracking which users are connected to the server.
   - **Room Membership:** Knowing which users are in which rooms.
   - **Typing Indicators:** Temporarily tracking and broadcasting when a user is actively composing a message in a specific room.
5. **State Querying:** Providing answers to questions like "What rooms exist?", "How many users are in room 'general'?", and "What are the last 20 messages in room 'lobby'?" (the latter via the Persistence layer).

Its scope ends at the boundary of the `ClientSession` and the `WebSocket`. It does not handle the raw WebSocket protocol, HTTP upgrades, or individual socket events. Instead, it operates on logical constructs (`connectionId`, `roomId`, `userId`) provided by the Connection Manager. It also defers permanent storage of messages and user credentials to the Persistence & Authentication Service, with which it collaborates.

Mental Model: The Publishing House and Subscribers

A helpful analogy for understanding this component is a **Publishing House (or Magazine)** and its **Subscribers**.

- **Each Chat Room is a distinct Magazine** (e.g., "Tech News", "Sports Talk"). The magazine has a title (`roomId`) and a list of current subscribers.
- **Users are Subscribers.** When a user joins a room, they subscribe to that magazine. They will receive every new issue (message) published to it.
- **Sending a Message is Publishing an Issue.** When a user sends a chat message to a room, they are publishing a new article to that magazine. The publishing house (Room Manager) immediately prints and mails a copy to every current subscriber of that magazine.
- **Typing Indicators are "Writer at Work" Signs.** If a subscriber starts drafting a letter to the editor (typing a message), the publishing house can put up a small sign in the magazine's office window: "John is writing...". This sign is visible to all other subscribers of that magazine.
- **The Connection Manager is the Postal Service.** It doesn't care about the content of the magazines; its job is to reliably deliver envelopes (WebSocket frames) between the publishing house and the individual subscribers' mailboxes (client browsers).

This model clarifies the decoupling: the Room Manager decides *what* to send and *to whom*, while the Connection Manager handles the *how* of the actual delivery.

Interface: Core Methods and Events

The Room Manager exposes a programmatic interface used primarily by the Connection Manager's message handlers. The following table details its key methods.

Method	Parameters	Returns	Description & Side Effects
`joinRoom`	`connectionId: string`, `roomId: string`, `userId: string`	`Promise<Room>`	Adds the user's `connectionId` to the room's `memberIds` set. Fetches recent message history from persistence. Broadcasts an `EVENT_USER_JOINED` message to all other members of the room. Returns the updated `Room` object.
`leaveRoom`	`connectionId: string`, `roomId: string`	`void`	Removes the `connectionId` from the room's `memberIds` set. If the room becomes empty, it may be scheduled for cleanup (see ADR). Broadcasts an `EVENT_USER_LEFT` message to the remaining members.
`leaveAllRooms`	`connectionId: string`	`void`	Removes the given `connectionId` from every room's membership. Typically called during connection cleanup (user disconnect).
`broadcastToRoom`	`roomId: string`, `message: WebSocketMessage`, `excludeConnectionId?: string`	`void`	Retrieves the set of `memberIds` for the given `roomId`. Iterates through each `connectionId`, and for each one that is not the `excludeConnectionId`, calls `ConnectionManager.sendToClient(connectionId, message)`.
`getRoomList`	-	`Array<{id: string, name: string, memberCount: number}>`	Returns a snapshot of all active rooms and their current member counts. Used for room listing UI.
`setUserTyping`	`connectionId: string`, `roomId: string`, `isTyping: boolean`	`void`	Updates an internal map tracking which users are typing in which rooms. Broadcasts an `EVENT_USER_TYPING` message to all other members of the room with the user's status. Manages a timeout to automatically clear the typing state after inactivity (e.g., 3 seconds).
`getRoomMembers`	`roomId: string`	`Set<string>` (connectionIds)	Returns the current set of `connectionId`s for a given room. Used for targeted operations and presence updates.

Key Internal Events: The Room Manager also listens for events from the Connection Manager, primarily `connectionClosed(connectionId)`. Upon receiving this, it must call `leaveAllRooms(connectionId)` to ensure state consistency.

ADR: In-Memory vs. External Room State

Decision: Store Room and Membership State In-Memory

- **Context:** The Room Manager needs fast, low-latency access to room membership data to route messages and track presence. This data is highly volatile (changes with every join/leave) and is only relevant to the currently running server instance. We must choose where to store this operational state.
- **Options Considered:**
  1. **In-Memory Data Structures:** Store `Room` objects (with `memberIds` sets) in a plain JavaScript `Map` within the Room Manager's process memory.
  2. **External Shared Database:** Store room membership in a fast, external data store like Redis (key: `room:<id>:members`, value: set of `userId` or `connectionId`).
- **Decision:** Use **In-Memory Data Structures** for the learning implementation.
- **Rationale:** This project has **intermediate** difficulty and focuses on learning WebSocket patterns, not distributed systems. Using in-memory structures drastically simplifies the code, eliminates network latency for membership checks, and requires no additional infrastructure (Redis). The primary con—lack of shared state across multiple servers—is acceptable because the non-goals explicitly exclude horizontal scaling for this version. It allows learners to focus on the core algorithms of room management without the complexity of distributed state synchronization.
- **Consequences:**
  - **Positive:** Maximum performance for read/write operations, simple implementation, no external dependencies.
  - **Negative:** State is lost on server restart. All rooms and membership information vanish. This is acceptable for a learning project, as clients will reconnect and re-join rooms. It also means the system cannot run in a multi-server (load-balanced) configuration without a major redesign.

Option	Pros	Cons	Chosen?
In-Memory	1. **Extremely low latency** (no network calls). 2. **Simple implementation** (native `Map` and `Set`). 3. **No external dependencies** to set up or manage.	1. **State is ephemeral** (lost on server crash/restart). 2. **Does not scale** beyond a single server process. 3. Harder to inspect/debug state externally.	**Yes** – Aligns with project's learning focus and non-goals.
External DB (e.g., Redis)	1. **State survives server restarts.** 2. **Enables horizontal scaling** (multiple chat servers can share state). 3. State can be inspected with external tools.	1. **Adds complexity** (requires Redis setup, connection pooling, serialization). 2. **Introduces network latency** for every membership operation. 3. **New failure modes** (Redis downtime, network partitions).	No – Overcomplicates the core learning objectives.

Common Pitfalls

This component has several subtle traps that can lead to confusing bugs.

⚠️ Pitfall 1: Accidental Broadcast to Sender (Echo)

- **Description:** When `broadcastToRoom` is called without providing the `excludeConnectionId` parameter, the user who sent the message receives their own message back via the WebSocket. This creates a confusing "echo" in their own client and often leads to duplicate message display.
- **Why it's wrong:** It wastes bandwidth and processing, and violates the typical user expectation in a chat app (you see your own message in the UI because your client rendered it optimistically on send, not because the server echoed it back).
- **Fix: Always pass the sender's `connectionId` as the `excludeConnectionId` argument** when broadcasting a message they originated. The `broadcastToRoom` method must explicitly skip this connection.

⚠️ Pitfall 2: Zombie Rooms (Memory Leak)

- **Description:** When the last user leaves a room, the `Room` object and its `memberIds` Set remain in the server's memory indefinitely. If users dynamically create many rooms (e.g., `project-abc`, `temp-meeting`), the server's memory will slowly fill with empty, unused rooms.
- **Why it's wrong:** It causes a memory leak, which can eventually crash the server under sustained use.
- **Fix:** Implement a cleanup mechanism. The `leaveRoom` method should check if the room's `memberIds` set is empty after removal. If it is, start a **configurable timeout** (e.g., 5 minutes). If no one joins the room before the timeout elapses, delete the `Room` object from the in-memory map. This allows for temporary empty rooms without permanent accumulation.

⚠️ Pitfall 3: Unsanitized Room Names

- **Description:** Accepting arbitrary user input (the desired room name) as a room identifier without validation can lead to security and stability issues. A user could create a room named `../../../etc/passwd` or `</script><script>alert('xss')</script>`.
- **Why it's wrong:** If the room name is used in file paths, database queries, or reflected in client HTML, it could enable path traversal, injection attacks, or cross-site scripting (XSS).
- **Fix: Sanitize and normalize room names.** Convert to a safe internal `roomId`. For example:
  1. Trim whitespace.
  2. Convert to lowercase.
  3. Replace any non-alphanumeric characters (except hyphens/underscores) with a hyphen.
  4. Enforce a reasonable length limit (e.g., 50 characters).
  5. Use this normalized string as the internal `roomId`. The original `name` can be stored for display but never used for logic.

⚠️ Pitfall 4: Race Condition in Typing Indicators

- **Description:** When a user stops typing, the client sends a `typing: false` event. If the user quickly starts typing again, the server might process the events out of order (due to network latency) resulting in the final state being `false` even though the user is actively typing.
- **Why it's wrong:** The typing indicator for that user disappears prematurely, confusing other chat participants.
- **Fix:** Use a **last-write-wins approach with a sequence number or timestamp.** Attach a monotonic increasing number or the client's timestamp to each typing event. The server only updates the typing state if the incoming event's number is greater than the one currently stored for that user/room. Alternatively, use a **debounce pattern on the server:** each `typing: true` event resets a user-specific timeout. The `typing: false` broadcast is only sent when that timeout fires without being reset.

⚠️ Pitfall 5: Not Validating Membership Before Broadcast

- **Description:** The `broadcastToRoom` method blindly retrieves a list of `connectionId`s and attempts to send to each. If a user has just disconnected, their `connectionId` may still be in the set, causing `ConnectionManager.sendToClient` to fail or log an error.
- **Why it's wrong:** It creates unnecessary error noise and wasted cycles. The room membership state (`memberIds`) becomes slightly stale.

- **Fix: The Connection Manager must be the source of truth for active connections.** When `leaveAllRooms` is called (during disconnect cleanup), it removes the `connectionId` from all room sets. This ensures the membership set is accurate before the next broadcast cycle. The `broadcastToRoom` method can then iterate with confidence. Implement this as a synchronous call from the Connection Manager's `connectionClosed` handler to the Room Manager's `leaveAllRooms`.

Implementation Guidance

This guidance provides the skeletal structure for the Room Manager component in JavaScript (Node.js), following the design outlined above.

A. Technology Recommendations

Component	Simple Option (Learning)	Advanced Option (Production)
In-Memory Store	JavaScript `Map` and `Set`	Redis with Pub/Sub for distributed state & messaging
Message Routing	Iterative loop over connection IDs	Topic-based Pub/Sub (Redis, NATS, Kafka)
Presence Tracking	In-memory maps with timeouts	Redis Sorted Sets with heartbeats

B. Recommended File/Module Structure

```
real-time-chat/
├── server/
│   ├── index.js                    # Main server entry point
│   ├── connectionManager.js        # From Milestone 1
│   ├── roomManager.js              # This component (NEW)
│   ├── authService.js              # For Milestone 4
│   ├── messageStore.js             # For Milestone 4
│   └── utils/
│       ├── validation.js           # parseAndValidateWebSocketMessage, etc.
│       └── messageFactory.js       # createServerMessage, createChatMessage
└── client/                         # Frontend HTML/CSS/JS
```

C. Infrastructure Starter Code

The following is a complete, usable utility for creating standardized server messages. Place this in `server/utils/messageFactory.js`.

JAVASCRIPT

```javascript
// server/utils/messageFactory.js

/**
 * Creates a properly formatted WebSocketMessage for server-originated events.
 * @param {string} type - The event type (e.g., EVENT_CHAT_MESSAGE).
 * @param {any} payload - The data payload for the event.
 * @returns {WebSocketMessage} A structured message object ready for JSON.stringify.
 */
function createServerMessage(type, payload) {
    return {
        type,
        payload,
        timestamp: new Date().toISOString() // Server's authoritative timestamp
    };
}

/**
 * Creates a new ChatMessage object with a server-assigned timestamp.
 * @param {string} sender - The username of the sender.
 * @param {string} content - The text content of the message.
 * @param {string} roomId - The ID of the room where the message was sent.
 * @returns {ChatMessage} A new ChatMessage object.
 */
function createChatMessage(sender, content, roomId) {
    return {
        sender,
        content,
        timestamp: new Date().toISOString(),
        roomId
    };
}

/**
 * Creates a new Room object.
 * @param {string} name - The display name of the room.
 * @returns {Room} A new Room object with a generated ID.
 */
function createRoom(name) {
    // Simple ID generation: sanitize name and add random suffix for uniqueness
    const baseId = name.toLowerCase().trim().replace(/[^a-z0-9-_]/g, '-');
    const randomSuffix = Math.random().toString(36).substring(2, 8);
    const id = `${baseId}-${randomSuffix}`;
```

```
    return {
        id,
        name,
        createdAt: new Date(),
        memberIds: new Set() // Tracks connectionIds, not userIds
    };
}

module.exports = {
    createServerMessage,
    createChatMessage,
    createRoom
};
```

D. Core Logic Skeleton Code

The main Room Manager class skeleton. Place this in `server/roomManager.js`. The TODO comments map directly to the algorithmic steps described in the design.

JAVASCRIPT

```javascript
// server/roomManager.js

const { createServerMessage } = require('./utils/messageFactory');

const { EVENT_USER_JOINED, EVENT_USER_LEFT, EVENT_USER_TYPING } = require('./constants');


class RoomManager {

    constructor(connectionManager, messageStore) {

        // In-memory store: Map<roomId, Room>

        this.rooms = new Map();

        // In-memory store: Map<connectionId, Set<roomId>> (reverse lookup)

        this.connectionRooms = new Map();

        // Typing state: Map<roomId, Map<userId, {isTyping: boolean, timeoutId: NodeJS.Timeout}>>

        this.typingState = new Map();


        // Dependencies

        this.connectionManager = connectionManager; // To send messages to clients

        this.messageStore = messageStore; // To fetch history (optional for Milestone 3, required for 4)

    }


    /**

     * Adds a connection to a room, fetches history, and notifies others.

     * @param {string} connectionId

     * @param {string} roomId

     * @param {string} userId

     * @returns {Promise<Room>} The joined room object.

     */

    async joinRoom(connectionId, roomId, userId) {

        // TODO 1: Check if the room exists. If not, create it using createRoom(roomId).

        // TODO 2: Retrieve the Room object from this.rooms.

        // TODO 3: Check if the user (connectionId) is already a member. If so, you may want to return early or handle re-join.

        // TODO 4: Add the connectionId to the room's memberIds Set.

        // TODO 5: Update the reverse lookup (this.connectionRooms): add this roomId to the set for this connectionId.

        // TODO 6: [For Milestone 4] Fetch recent message history for this room from this.messageStore (e.g., last 50 messages).

        // TODO 7: Broadcast an EVENT_USER_JOINED message to all OTHER members of the room.

        //         Use createServerMessage(EVENT_USER_JOINED, { userId, roomId }) and broadcastToRoom (excluding the new joiner).

        // TODO 8: Return the Room object (and optionally the message history to be sent directly to the joiner).

    }


    /**

     * Removes a connection from a specific room and notifies others.

     * @param {string} connectionId

     * @param {string} roomId

     */
```

```
    leaveRoom(connectionId, roomId) {

        // TODO 1: Retrieve the Room object from this.rooms. If room doesn't exist, return.

        // TODO 2: Remove the connectionId from the room's memberIds Set.

        // TODO 3: Update the reverse lookup: remove this roomId from the set for this connectionId.

        // TODO 4: Broadcast an EVENT_USER_LEFT message to the REMAINING members of the room.

        // TODO 5: Check if the room's memberIds set is now empty. If yes, schedule this room for cleanup (start a timer to delete it after X minutes).

    }


    /**

     * Removes a connection from all rooms it belongs to.

     * Called when a WebSocket connection is closed.

     * @param {string} connectionId

     */

    leaveAllRooms(connectionId) {

        // TODO 1: Get the set of roomIds this connection is in from this.connectionRooms. If empty, return.

        // TODO 2: For each roomId in the set, call this.leaveRoom(connectionId, roomId).

        // TODO 3: Delete the entry for this connectionId from this.connectionRooms.

        // TODO 4: Clear any typing indicators for this connection/user across all rooms.

    }


    /**

     * Sends a message to all members of a room, excluding an optional connection.

     * @param {string} roomId

     * @param {WebSocketMessage} message

     * @param {string} [excludeConnectionId] - The connectionId to exclude (usually the sender).

     */

    broadcastToRoom(roomId, message, excludeConnectionId) {

        // TODO 1: Retrieve the Room object. If room doesn't exist, log a warning and return.

        // TODO 2: Iterate over the room.memberIds Set.

        // TODO 3: For each memberConnectionId, if it is NOT equal to excludeConnectionId, call this.connectionManager.sendToClient(memberConnectionId, message).

        //         Hint: Check the connection manager's method signature.

    }


    /**

     * Updates and broadcasts a user's typing status in a room.

     * @param {string} connectionId

     * @param {string} roomId

     * @param {boolean} isTyping

     */

    setUserTyping(connectionId, roomId, isTyping) {

        // TODO 1: Retrieve the userId for this connectionId from the connectionManager (you may need to expose a getSession method).
```

```
        // TODO 2: Get or create the typing state map for this room (from this.typingState).
        // TODO 3: Clear any existing timeout for this user in this room to prevent premature "stopped typing" messages.
        // TODO 4: If isTyping is true:
        //          a. Store the new state (isTyping: true) and set a new timeout for 3000ms.
        //          b. When the timeout fires, automatically call setUserTyping again with isTyping: false for this user/room.
        //          c. Broadcast an EVENT_USER_TYPING message to all OTHER room members with { userId, roomId, isTyping: true }.
        // TODO 5: If isTyping is false:
        //          a. Clear the timeout and remove the user's state from the room's map.
        //          b. If the room's map becomes empty, you can delete it from this.typingState.
        //          c. Broadcast an EVENT_USER_TYPING message with isTyping: false.
    }

    /**
     * Returns a list of all active rooms with their member counts.
     * @returns {Array<{id: string, name: string, memberCount: number}>}
     */
    getRoomList() {
        // TODO 1: Initialize an empty array `roomList`.
        // TODO 2: Iterate over this.rooms.values().
        // TODO 3: For each Room object, push an object with id, name, and memberCount (room.memberIds.size) to roomList.
        // TODO 4: Return roomList.
    }
}

module.exports = RoomManager;
```

E. Language-Specific Hints

- **Use `Set` for Membership:** The `Room.memberIds` should be a JavaScript `Set` for O(1) addition, deletion, and lookup.
- **Maps for Fast Lookup:** Use `Map` for `this.rooms` (keyed by `roomId`) and `this.connectionRooms` (keyed by `connectionId`). They preserve insertion order and are more efficient than plain objects for frequent additions/deletions.
- **Cleaning Timeouts:** Always store the `timeoutId` returned by `setTimeout` when scheduling room cleanup or typing indicator expiration. Clear it with `clearTimeout(timeoutId)` if the room becomes non-empty again or the user starts typing again to prevent memory leaks.
- **Error Handling in Broadcast:** Wrap the `this.connectionManager.sendToClient` call in a try-catch block or ensure the connection manager handles closed sockets gracefully and doesn't throw.

F. Milestone Checkpoint

After implementing the Room Manager and integrating it with the Connection Manager from Milestone 1, you should be able to test core chat functionality.

Expected Behavior (Milestone 2 & 3):

1. Start your server (`node server/index.js`).
2. Open two different browser tabs to your client page.
3. In Tab 1, join a room (e.g., "general"). You should see a system message confirming your join.
4. In Tab 2, join the same room "general". Tab 1 should now receive a "user joined" notification.
5. Type a message in Tab 2 and send it. The message should appear in Tab 1's chat log, but **not** a second time in Tab 2 (no echo). The message should have a timestamp, sender name, and content.
6. Start typing in Tab 1. After a brief delay, Tab 2 should show a "User is typing..." indicator, which disappears when you send the message or stop typing for a few seconds.
7. In a third tab, join a different room (e.g., "random"). Messages sent in "general" should **not** appear in this tab, demonstrating room isolation.

Commands to Verify:

- Observe server logs. You should see log entries for `joinRoom`, `broadcastToRoom`, and `leaveRoom` operations.
- Check that the number of active connections and room counts are logged correctly.

Signs of Trouble:

- **Messages appear duplicated in sender's tab:** You are not excluding the sender in `broadcastToRoom`.
- **"User joined" message appears for the user themselves:** The `EVENT_USER_JOINED` broadcast is not excluding the new joiner.
- **Typing indicator never disappears:** The timeout logic in `setUserTyping` is not correctly clearing or setting the state to false.
- **Server memory usage grows indefinitely as rooms are created/left:** The "zombie room" cleanup in `leaveRoom` is not implemented.

Component Design: Persistence & Authentication Service

> **Milestone(s):** Milestone 4 (User Authentication & Persistence)

This component provides the foundational services for **user identity management** and **message persistence**—two critical requirements for any production chat application. While the WebSocket Server and Room Manager handle real-time interactions, the Persistence & Authentication Service ensures that user identities are verified, conversations are preserved, and historical context is available when users reconnect or join rooms.

Responsibility and Scope

The Persistence & Authentication Service operates as two logically separate but closely collaborating subsystems:

Subsystem	Primary Responsibilities	Key Data Managed
Authentication Service	1. User registration and credential storage 2. Session creation and validation during WebSocket connection establishment 3. User profile management (username, status) 4. Session cleanup (expired/inactive sessions)	- `User` records (username, password hash) - Session tokens or cookies - Authentication state per connection
Message Persistence Store	1. Storing `ChatMessage` objects with metadata (sender, room, timestamp) 2. Retrieving message history with pagination support 3. Room message count and statistics 4. Optional message archival/cleanup policies	- `ChatMessage` objects indexed by room and timestamp - Message ID sequencing - Pagination cursors

The service interfaces with the **WebSocket Server** during connection establishment (for authentication) and with the **Room Manager** when messages need to be stored or retrieved. Its scope is deliberately bounded to identity and persistence concerns, leaving real-time routing and room state management to the Room Manager.

Mental Model: The Library Archivist and Security Guard

To build intuition, imagine this service as two roles in a physical library:

> **The Security Guard (Authentication Service)** stands at the entrance, checking library cards (sessions) before allowing anyone inside. They maintain a registry of valid members (`User` records) and issue temporary passes (sessions) that expire after a period of inactivity. When someone presents an expired or forged pass, the guard denies entry and logs the attempt.

> **The Library Archivist (Message Persistence Store)** works in the stacks, meticulously cataloging every conversation that occurs in the library's reading rooms. They organize messages by room (like books by shelf), maintain a chronological index, and can retrieve specific conversations when patrons request them. The archivist doesn't participate in the conversations but ensures they're preserved for future reference.

This mental model clarifies the separation of concerns: authentication is a **gatekeeping** function performed at entry points, while persistence is a **record-keeping** function that operates continuously in the background.

ADR: When to Authenticate - HTTP vs. WebSocket

> **Decision: Authenticate During HTTP Upgrade, Not Via WebSocket Messages**
>
> **Context**: WebSocket connections begin as HTTP requests that "upgrade" to the WebSocket protocol. We need to verify user identity before allowing them to establish a persistent connection and access chat features, but we have multiple timing options for performing this verification.
>
> **Options Considered**:
>
> 1. **HTTP Upgrade Authentication**: Validate credentials/session during the initial HTTP upgrade request before establishing the WebSocket connection.
> 2. **Custom WebSocket Authentication Message**: Accept all WebSocket connections, then require clients to send a special authentication message as their first action.
> 3. **Hybrid Approach**: Use HTTP for initial handshake with tokens, then allow reauthentication via WebSocket messages for session renewal.
>
> **Decision**: We chose Option 1 (HTTP Upgrade Authentication) as the primary authentication mechanism.
>
> **Rationale**:
>
> - **Security**: Rejecting unauthenticated connections at the HTTP layer prevents resource consumption by unauthorized clients. The WebSocket connection is never established if authentication fails.
> - **Simplicity**: The authentication logic runs once at connection time, simplifying subsequent message handling (no need to check auth state on every message).
> - **Established Pattern**: This aligns with how production WebSocket services (like Socket.IO with middleware) typically handle authentication using HTTP cookies or tokens.
> - **Early Failure**: Clients receive immediate feedback (HTTP 401/403) rather than establishing a connection only to be disconnected later.
>
> **Consequences**:
>
> - **Positive**: Reduced server load from unauthorized connections, cleaner client code (auth handled during connection setup), immediate user feedback on auth failures.
> - **Negative**: Requires session management via HTTP cookies or tokens, cannot easily change user identity without reconnecting, slightly more complex upgrade handler implementation.
> - **Mitigation**: We'll implement a simple token-based system where clients pass an authentication token in the upgrade request headers.

Option	Pros	Cons	Why Chosen?
HTTP Upgrade Authentication	- Early rejection saves resources - Single authentication point - Aligns with security best practices	- Requires token/cookie management - Cannot reauthenticate without reconnecting	**CHOSEN**: Provides security and simplicity for learning context
WebSocket Message Authentication	- Flexible reauthentication - Can support multiple auth methods	- Unauthenticated connections consume resources - Must validate auth on every message initially - Complex state management	Rejected due to security concerns and complexity
Hybrid Approach	- Best of both worlds - Supports session renewal	- Most complex implementation - Overkill for learning context	Rejected as overly complex for our needs

Common Pitfalls: Unbounded History and Session Leaks

⚠️ Pitfall: Loading Entire Message History Without Pagination

Description: When a user joins a room, the server attempts to load all historical messages from the database, which could be thousands or millions of records.

Why It's Wrong: This causes severe performance issues:

- High memory consumption on both server and client
- Long loading times blocking other operations
- Potential denial of service if many users join simultaneously
- Network bandwidth exhaustion sending massive payloads

How to Fix: Implement **pagination** with a reasonable default limit (e.g., 50 most recent messages). Use cursor-based pagination where the client can request older messages on demand. Store only the necessary fields in memory, not entire message objects.

⚠️ Pitfall: Never Expiring Sessions or Cleaning Up Stale Data

Description: Sessions (`ClientSession` objects) remain in memory indefinitely after users disconnect, and old messages accumulate in the database without cleanup.

Why It's Wrong: This leads to resource exhaustion and stale state:

- Memory leaks as disconnected sessions accumulate
- Database bloat affecting query performance
- Security risk from indefinitely valid session tokens
- Incorrect presence information if stale sessions aren't cleaned

How to Fix: Implement **TTL (Time-To-Live)** policies:

1. Sessions: Remove `ClientSession` after WebSocket disconnection + short grace period
2. Authentication tokens: Set expiration times (e.g., 24 hours)
3. Messages: Consider archival or deletion policies for very old messages
4. Use heartbeat mechanisms to detect and clean up "zombie" connections

---

⚠️ Pitfall: Storing Passwords in Plain Text

Description: User passwords are stored as plain text strings in the database rather than as cryptographic hashes.

Why It's Wrong: This creates catastrophic security vulnerabilities:

- Database compromise exposes all user passwords
- Users who reuse passwords across sites are at risk
- Violates security best practices and potentially regulations

How to Fix: Always hash passwords using a **key derivation function** like bcrypt, scrypt, or Argon2. Store only the hash, never the plaintext. During authentication, hash the provided password and compare it to the stored hash.

⚠️ Pitfall: No Validation or Sanitization of Persistent Data

Description: User-generated content (messages, room names, usernames) is stored directly without validation, sanitization, or encoding.

Why It's Wrong: This enables multiple attack vectors:

- **XSS (Cross-Site Scripting)**: Malicious scripts in messages execute in other users' browsers
- **SQL/NoSQL Injection**: If using string concatenation for queries
- **Data corruption**: Invalid data breaks application logic
- **Storage abuse**: Extremely large messages consume excessive resources

How to Fix: Implement **layered validation**:

1. **Client-side**: Basic validation for user experience
2. **Server-side before storage**:
   - Length limits (enforce `MAX_MESSAGE_LENGTH`)
   - Character encoding/escaping
   - Content type validation
   - Sanitize HTML/script tags from messages
3. **Database layer**: Use parameterized queries to prevent injection

Interface: Core Methods and Events

The Authentication Service and Message Store expose the following interfaces to other components:

Authentication Service Interface

Method	Parameters	Returns	Description
`authenticateConnection(request)`	`request` : HTTP Upgrade request object	`{ success: boolean, user: User | null, error: string | null }`	Validates credentials/token from HTTP upgrade headers. Returns authenticated `User` object or error.
`registerUser(username, password)`	`username` : string, `password` : string	`Promise<{ success: boolean, user: User | null, error: string | null }>`	Creates new user with hashed password. Validates uniqueness, password strength.
`validateSession(sessionToken)`	`sessionToken` : string	`Promise<{ valid: boolean, user: User | null }>`	Validates existing session token without creating new connection. Used for reconnection.
`createSession(userId, connectionInfo)`	`userId` : string, `connectionInfo` : { ip, userAgent }	`Promise<string>` (sessionToken)	Creates a new session for an authenticated user, returning a token for future validation.
`invalidateSession(sessionToken)`	`sessionToken` : string	`Promise<boolean>`	Removes session, effectively logging out the user from that device/connection.

Message Persistence Store Interface

Method	Parameters	Returns	Description
`storeMessage(message)`	`message` : `ChatMessage` object	`Promise<ChatMessage>` (with assigned ID)	Stores a chat message with timestamp, assigns unique ID, indexes by room and timestamp.
`getMessagesForRoom(roomId, options)`	`roomId` : string, `options` : { limit: number, before: timestamp | null }	`Promise<ChatMessage[]>`	Retrieves messages for a room with pagination. `before` parameter enables fetching older messages.
`getRecentMessagesForRoom(roomId, limit)`	`roomId` : string, `limit` : number (default: 50)	`Promise<ChatMessage[]>`	Convenience method for getting most recent messages when joining a room.
`getMessageCountForRoom(roomId)`	`roomId` : string	`Promise<number>`	Returns total message count in a room (for statistics or UI indicators).
`deleteMessagesOlderThan(timestamp)`	`timestamp` : ISO date string	`Promise<number>` (deleted count)	Cleanup method for removing very old messages (optional, for maintenance).

Data Flow: Authentication During Connection Establishment

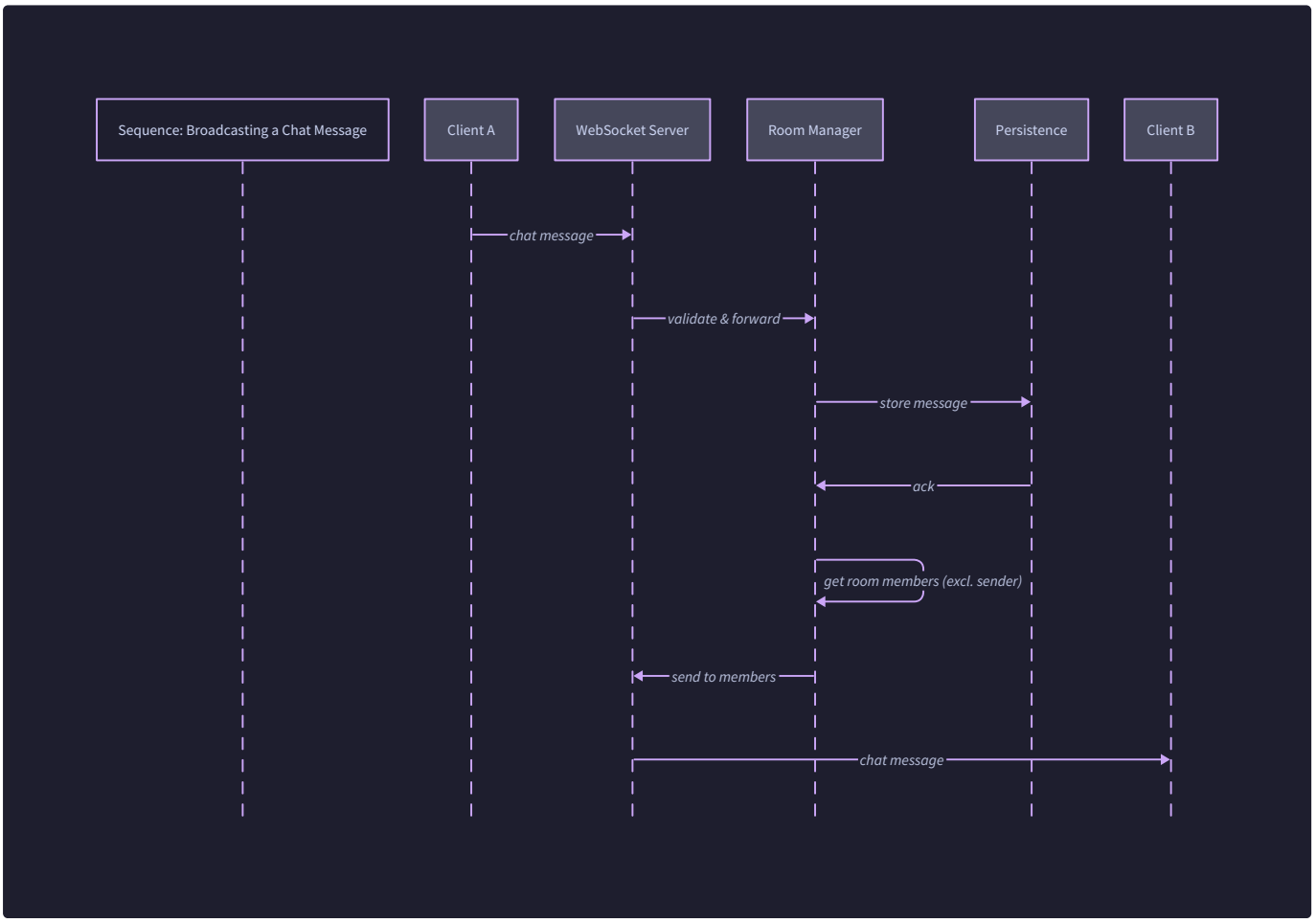
The authentication process integrates with the WebSocket connection lifecycle described in previous sections:

1. **Client initiates connection**: Browser JavaScript attempts to establish WebSocket connection with server endpoint.
2. **HTTP Upgrade request**: The WebSocket handshake begins as an HTTP GET request with `Upgrade: websocket` header.
3. **Authentication interceptor**: The WebSocket Server's HTTP upgrade handler extracts authentication credentials (typically from `Authorization` header or cookie).
4. **Credential validation**: The Authentication Service validates the token/session:
    - **Valid session**: Proceeds to step 5
    - **Invalid/expired session**: Returns HTTP 401 Unauthorized, connection fails
    - **No credentials**: Returns HTTP 403 Forbidden, connection fails
5. **User lookup**: If credentials are valid, the Authentication Service retrieves the corresponding `User` object.
6. **Session creation**: A new `ClientSession` is created linking the WebSocket connection to the authenticated user.
7. **WebSocket connection established**: HTTP 101 Switching Protocols is sent, WebSocket protocol begins.
8. **Initialization message**: Server sends `connection_established` event with user info and available rooms.

This flow ensures that **every active WebSocket connection has an associated authenticated user**, simplifying subsequent authorization checks (e.g., "does this user have permission to post in this room?").

Data Flow: Message Persistence

When a chat message flows through the system (as shown in



), persistence occurs at a specific point:

1. **Message reception**: `ConnectionManager.handleMessage()` receives a `chat_message` event from a client.
2. **Validation and processing**: The message is validated, a `ChatMessage` object is created with server timestamp.
3. **Storage before broadcast**: Before broadcasting to room members, the Room Manager calls `MessageStore.storeMessage(message)` to persist it.
4. **Synchronous storage**: The message is written to the database. In our learning implementation, we'll wait for confirmation before proceeding.
5. **Broadcast**: Once storage is confirmed, the message is broadcast to room members.

> **Design Insight**: We store **before** broadcasting to ensure no message is delivered to users without being persisted. This guarantees that if the server crashes between broadcast and storage, the message won't be lost entirely (it won't have been seen by anyone). This is a **safety-first** approach suitable for chat applications where message loss is more critical than minimal latency.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option (Learning)	Advanced Option (Production)
Authentication Storage	In-memory JavaScript Map	Redis (sessions) + PostgreSQL (users)
Message Storage	In-memory array with periodic file dump	PostgreSQL with JSONB or dedicated message DB
Password Hashing	bcrypt library (npm `bcrypt`)	bcrypt with appropriate work factor
Session Tokens	JSON Web Tokens (JWT)	JWT with refresh token rotation

B. Recommended File/Module Structure

```
real-time-chat/
├── server/
│   ├── index.js                  # Main server entry point
│   ├── connection-manager.js     # From Milestone 1
│   ├── room-manager.js           # From Milestones 2-3
│   ├── auth-service.js           # THIS COMPONENT: Authentication logic
│   ├── message-store.js          # THIS COMPONENT: Message persistence
│   ├── models.js                 # Shared type definitions (User, ChatMessage, etc.)
│   └── utils.js                  # Helper functions (hashing, validation)
├── client/
│   └── public/                   # HTML, CSS, client JavaScript
└── package.json
```

C. Infrastructure Starter Code: Simple In-Memory Message Store

Here's a complete, working in-memory message store suitable for learning and development. This implements the `MessageStore` interface with basic pagination:

JAVASCRIPT

```javascript
// server/message-store.js

/**
 * Simple in-memory message store with periodic persistence to file.
 * For learning purposes only - not suitable for production.
 */
const fs = require('fs').promises;
const path = require('path');

class InMemoryMessageStore {
  constructor(persistenceFile = './message-store.json', persistIntervalMs = 30000) {
    this.messages = new Map(); // roomId -> Array<ChatMessage>
    this.persistenceFile = persistenceFile;
    this.persistIntervalMs = persistIntervalMs;
    this.nextMessageId = 1;

    // Load existing messages from file if it exists
    this.loadFromFile().catch(err => {
      console.warn('Could not load message store from file:', err.message);
    });

    // Set up periodic persistence
    this.persistenceInterval = setInterval(() => {
      this.persistToFile().catch(err => {
        console.error('Failed to persist messages:', err);
      });
    }, this.persistIntervalMs);
  }

  /**
   * Store a chat message
   * @param {ChatMessage} message - Must have roomId, sender, content, timestamp
   * @returns {Promise<ChatMessage>} - Message with assigned id
   */
  async storeMessage(message) {
    // Validate required fields
    if (!message.roomId || !message.sender || !message.content || !message.timestamp) {
      throw new Error('Message missing required fields');
    }

    // Assign unique ID
    const messageWithId = {
```

```
    ...message,
    id: this.nextMessageId++,
  };

  // Initialize room array if needed
  if (!this.messages.has(message.roomId)) {
    this.messages.set(message.roomId, []);
  }

  // Add to room's messages
  this.messages.get(message.roomId).push(messageWithId);

  return messageWithId;
}

/**
 * Get messages for a room with pagination
 * @param {string} roomId
 * @param {Object} options
 * @param {number} options.limit - Maximum messages to return
 * @param {string|null} options.before - ISO timestamp: return messages before this time
 * @returns {Promise<ChatMessage[]>}
 */
async getMessagesForRoom(roomId, options = {}) {
  const { limit = 50, before = null } = options;

  if (!this.messages.has(roomId)) {
    return [];
  }

  let roomMessages = this.messages.get(roomId);

  // Filter by 'before' timestamp if provided
  if (before) {
    roomMessages = roomMessages.filter(msg => msg.timestamp < before);
  }

  // Sort by timestamp descending (newest first) and take limit
  return roomMessages
    .sort((a, b) => new Date(b.timestamp) - new Date(a.timestamp))
    .slice(0, limit);
```

```
}

/**
 * Convenience method for getting recent messages
 * @param {string} roomId
 * @param {number} limit
 * @returns {Promise<ChatMessage[]>}
 */
async getRecentMessagesForRoom(roomId, limit = 50) {
  return this.getMessagesForRoom(roomId, { limit });
}

/**
 * Get message count for a room
 * @param {string} roomId
 * @returns {Promise<number>}
 */
async getMessageCountForRoom(roomId) {
  return this.messages.has(roomId) ? this.messages.get(roomId).length : 0;
}

/**
 * Save messages to disk
 * @private
 */
async persistToFile() {
  const data = {
    messages: Object.fromEntries(this.messages),
    nextMessageId: this.nextMessageId,
    persistedAt: new Date().toISOString(),
  };

  await fs.writeFile(
    this.persistenceFile,
    JSON.stringify(data, null, 2),
    'utf-8'
  );
}

/**
 * Load messages from disk
 * @private
```

```
 */
async loadFromFile() {
  try {
    const data = await fs.readFile(this.persistenceFile, 'utf-8');
    const parsed = JSON.parse(data);

    this.messages = new Map(Object.entries(parsed.messages || {}));
    this.nextMessageId = parsed.nextMessageId || 1;

    console.log(`Loaded ${Array.from(this.messages.values()).flat().length} messages from disk`);
  } catch (err) {
    if (err.code !== 'ENOENT') {
      throw err; // Re-throw if it's not "file not found"
    }
  }
}

/**
 * Cleanup old messages (optional)
 * @param {string} cutoffDate - ISO timestamp
 * @returns {Promise<number>} - Number of messages deleted
 */
async deleteMessagesOlderThan(cutoffDate) {
  let totalDeleted = 0;
  const cutoff = new Date(cutoffDate);

  for (const [roomId, messages] of this.messages.entries()) {
    const originalLength = messages.length;
    const filtered = messages.filter(msg => new Date(msg.timestamp) >= cutoff);

    if (filtered.length !== originalLength) {
      this.messages.set(roomId, filtered);
      totalDeleted += (originalLength - filtered.length);
    }
  }

  return totalDeleted;
}

/**
 * Graceful shutdown - persist before exiting
```

```
   */
  async shutdown() {
    clearInterval(this.persistenceInterval);
    await this.persistToFile();
  }
}

// Export a singleton instance
module.exports = new InMemoryMessageStore();
```

D. Core Logic Skeleton: Authentication Service

```javascript
// server/auth-service.js

const bcrypt = require('bcrypt');

const { v4: uuidv4 } = require('uuid');

const { User } = require('./models');


/**

 * Authentication Service responsible for user registration, login,

 * and session management.

 */

class AuthenticationService {

  constructor() {

    // In-memory stores for learning purposes

    this.users = new Map(); // username -> User object

    this.sessions = new Map(); // sessionToken -> { userId, createdAt, lastActivity }



    // Configuration

    this.sessionTTL = 24 * 60 * 60 * 1000; // 24 hours in milliseconds

    this.saltRounds = 10; // For bcrypt hashing



    // Initialize with a test user for development

    this.initializeTestUser();

  }


  /**

   * Authenticate a user during WebSocket upgrade

   * @param {http.IncomingMessage} request - HTTP upgrade request

   * @returns {Object} { success: boolean, user: User|null, error: string|null }

   */

  authenticateConnection(request) {

    // TODO 1: Extract authentication token from request headers

    // Look for 'Authorization' header with format 'Bearer <token>'

    // or check cookies if using cookie-based auth



    // TODO 2: If no token found, return { success: false, user: null, error: 'No credentials provided' }



    // TODO 3: Validate the session token using this.validateSession(token)



    // TODO 4: If session is valid, retrieve the User object from this.users



    // TODO 5: Update session's lastActivity timestamp
```

```
  // TODO 6: Return { success: true, user: userObject, error: null }

  // TODO 7: If any step fails, return appropriate error message
}

/**
 * Register a new user
 * @param {string} username
 * @param {string} plainPassword
 * @returns {Promise<Object>} { success: boolean, user: User|null, error: string|null }
 */
async registerUser(username, plainPassword) {
  // TODO 1: Validate username (length, characters, uniqueness)
  // Check if username already exists in this.users

  // TODO 2: Validate password strength (minimum length, complexity if desired)

  // TODO 3: Hash the password using bcrypt.hash(plainPassword, this.saltRounds)

  // TODO 4: Create new User object with username, passwordHash, createdAt

  // TODO 5: Store in this.users map

  // TODO 6: Return success with user object (EXCLUDE passwordHash from returned object)

  // TODO 7: Handle errors (duplicate username, hashing failure, etc.)
}

/**
 * Login existing user and create session
 * @param {string} username
 * @param {string} plainPassword
 * @returns {Promise<Object>} { success: boolean, sessionToken: string|null, error: string|null }
 */
async loginUser(username, plainPassword) {
  // TODO 1: Find user by username in this.users

  // TODO 2: If user not found, return { success: false, sessionToken: null, error: 'Invalid credentials' }

  // TODO 3: Compare provided password with stored hash using bcrypt.compare()
```

```
    // TODO 4: If password doesn't match, return same error as step 2 (security: don't reveal if user exists)

    // TODO 5: Generate a new session token (use uuidv4())

    // TODO 6: Store session in this.sessions with userId, createdAt, lastActivity

    // TODO 7: Return { success: true, sessionToken: token, error: null }
  }

  /**
   * Validate an existing session token
   * @param {string} sessionToken
   * @returns {Promise<Object>} { valid: boolean, user: User|null }
   */
  async validateSession(sessionToken) {
    // TODO 1: Look up session in this.sessions

    // TODO 2: If session not found, return { valid: false, user: null }

    // TODO 3: Check if session has expired (compare lastActivity + TTL to current time)

    // TODO 4: If expired, remove from this.sessions and return { valid: false, user: null }

    // TODO 5: If valid, update lastActivity timestamp

    // TODO 6: Retrieve user from this.users and return { valid: true, user: userObject }
  }

  /**
   * Create a session for an already-authenticated user
   * @param {string} userId
   * @param {Object} connectionInfo - { ip: string, userAgent: string }
   * @returns {Promise<string>} sessionToken
   */
  async createSession(userId, connectionInfo) {
    // TODO 1: Generate new session token (uuidv4)

    // TODO 2: Create session object with userId, createdAt, lastActivity, connectionInfo

    // TODO 3: Store in this.sessions
```

```
  // TODO 4: Return the session token
}

/**
 * Invalidate (logout) a session
 * @param {string} sessionToken
 * @returns {Promise<boolean>} true if session was found and removed
 */
async invalidateSession(sessionToken) {
  // TODO 1: Check if session exists in this.sessions

  // TODO 2: If exists, delete it and return true

  // TODO 3: If not found, return false
}

/**
 * Clean up expired sessions (should be called periodically)
 * @returns {number} count of sessions cleaned up
 */
cleanupExpiredSessions() {
  // TODO 1: Get current timestamp

  // TODO 2: Iterate through this.sessions

  // TODO 3: For each session, check if lastActivity + TTL < current time

  // TODO 4: Delete expired sessions

  // TODO 5: Return count of deleted sessions
}

/**
 * Initialize with a test user for development
 * @private
 */
async initializeTestUser() {
  const testUsername = 'testuser';
  const testPassword = 'password123';

  if (!this.users.has(testUsername)) {
```

```
      try {
        const passwordHash = await bcrypt.hash(testPassword, this.saltRounds);
        this.users.set(testUsername, {
          username: testUsername,
          passwordHash,
          createdAt: new Date().toISOString(),
        });
        console.log(`Test user created: ${testUsername} / ${testPassword}`);
      } catch (err) {
        console.error('Failed to create test user:', err);
      }
    }
  }
}

// Export a singleton instance
module.exports = new AuthenticationService();
```

E. Language-Specific Hints (JavaScript/Node.js)

- **Password Hashing**: Use the `bcrypt` npm package. Install with `npm install bcrypt`. Always use `async/await` with bcrypt as the hashing is CPU-intensive and blocking the event loop would hurt performance.
- **Session Tokens**: Use `uuid` package for generating secure random tokens: `npm install uuid`. Version 4 (random) UUIDs are sufficient for learning. In production, consider signed tokens (JWT) for stateless validation.
- **HTTP Header Parsing**: In the `authenticateConnection` method, access headers via `request.headers['authorization']`. Remember header names are lowercase in Node.js's parsed headers object.
- **Memory Management**: The in-memory stores will grow indefinitely. For a learning project, this is acceptable, but add a `/debug` endpoint that shows memory usage and allows manual cleanup.
- **Error Handling**: Always use `try/catch` around async operations with bcrypt and file I/O. Return user-friendly error messages but log detailed errors server-side.

F. Integration with WebSocket Server

Update your WebSocket server from Milestone 1 to integrate authentication:

JAVASCRIPT

```javascript
// In your main server file (server/index.js)

const authService = require('./auth-service');

const http = require('http');


const server = http.createServer();

const wss = new WebSocket.Server({ noServer: true });


server.on('upgrade', (request, socket, head) => {

  // Authenticate before upgrading to WebSocket

  const authResult = authService.authenticateConnection(request);



  if (!authResult.success) {

    socket.write('HTTP/1.1 401 Unauthorized\r\n\r\n');

    socket.destroy();

    return;

  }



  // Proceed with WebSocket upgrade for authenticated users

  wss.handleUpgrade(request, socket, head, (ws) => {

    // Create ClientSession linking ws to authenticated user

    const clientSession = createClientSession(

      ws,

      authResult.user.username,

      request.socket.remoteAddress,

      request.headers['user-agent']

    );



    // Store in ConnectionManager

    connectionManager.addClient(clientSession);



    // Send welcome message

    ws.send(JSON.stringify({

      type: 'connection_established',

      payload: {

        user: authResult.user.username,

        timestamp: new Date().toISOString()

      }

    }));

  });

});
```

G. Milestone Checkpoint for Authentication & Persistence

After implementing this component, verify your system works correctly:

1. **Start the server**: `node server/index.js`
2. **Test registration** (via a simple curl or test client):

```bash
curl -X POST http://localhost:3000/register \
  -H "Content-Type: application/json" \
  -d '{"username":"alice","password":"secure123"}'
```

Should return: `{"success":true,"user":{"username":"alice","createdAt":"..."}}`

3. **Test login**:

```bash
curl -X POST http://localhost:3000/login \
  -H "Content-Type: application/json" \
  -d '{"username":"alice","password":"secure123"}'
```

Should return a session token.

4. **Connect with WebSocket using authentication**:
   - Modify your client code to include the session token in WebSocket connection headers
   - Verify connection succeeds with authenticated user
   - Verify connection fails with invalid/no token
5. **Test message persistence**:
   - Send a chat message in a room
   - Disconnect and reconnect
   - Verify previous messages load when rejoining the room
   - Check that only recent messages load (not entire history)
6. **Verify session cleanup**:
   - Connect and authenticate
   - Disconnect client
   - Wait for session TTL + grace period
   - Try to reconnect with old token (should fail)

Expected Behavior: Users can register, login, establish authenticated WebSocket connections, and see message history when joining rooms. Sessions expire appropriately, and passwords are securely hashed.

Troubleshooting:

- "Registration fails with duplicate username": Check `users` Map for existing username
- "Login fails even with correct password": Verify bcrypt.compare() is working (hashes must match exactly)
- "Messages not loading on rejoin": Check `message-store.js` persistence file location and permissions
- "Session valid after disconnect": Ensure `connectionManager.removeClient()` calls session cleanup

---

Interactions and Data Flow

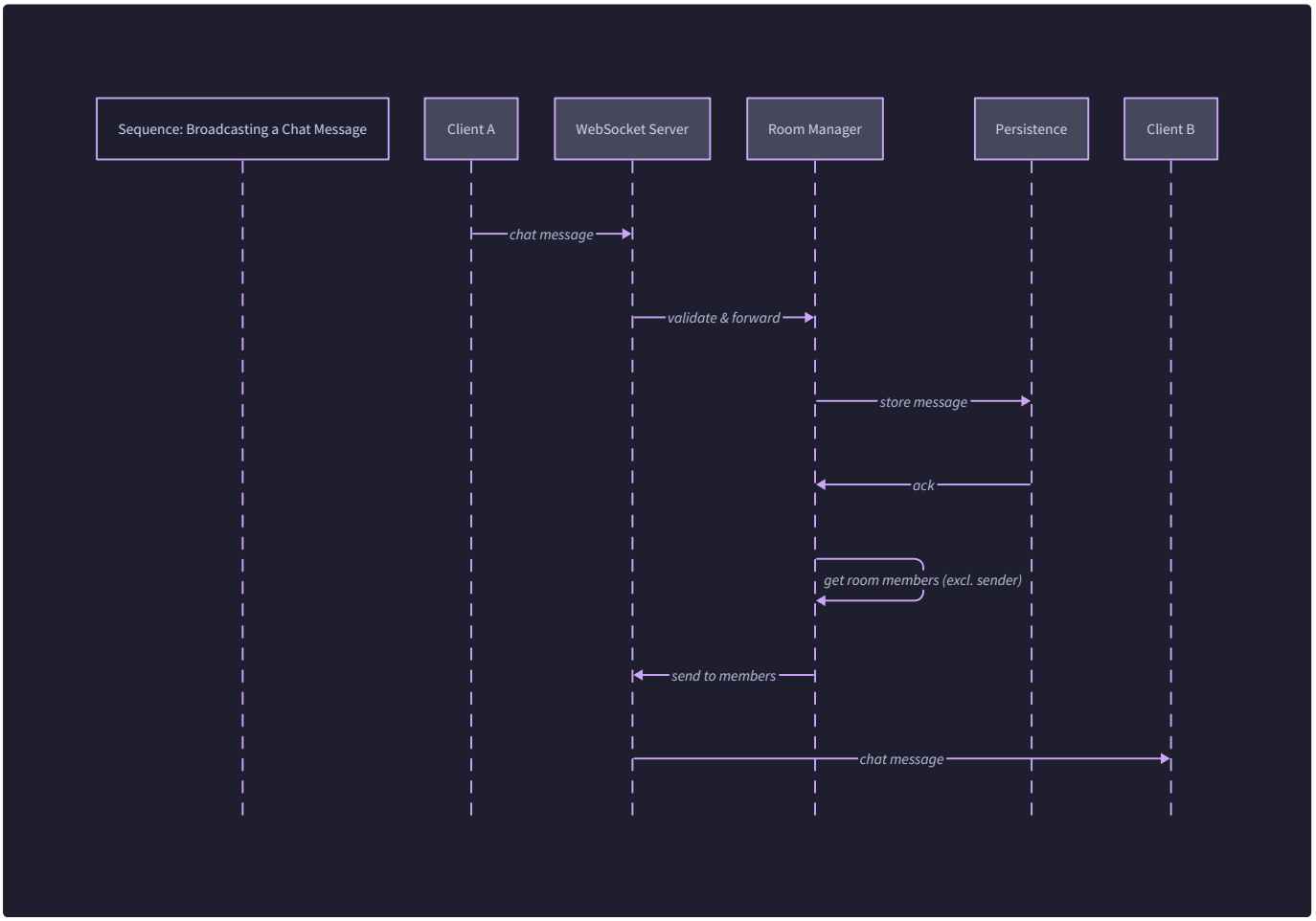
> **Milestone(s):** Milestone 2 (Message Broadcasting), Milestone 3 (Chat Rooms), Milestone 4 (User Authentication & Persistence) - for history loading and authentication

This section traces the concrete journeys through our chat system. Understanding these flows—the precise sequence of events, data transformations, and component interactions—is crucial for building a mental model of how the system behaves under real conditions. We'll follow two fundamental user stories: sending a message to a room, and joining a room to participate in conversation.

Sequence: Sending and Receiving a Message

Mental Model: The Party Conversation Imagine a lively party where people gather in different rooms. When someone speaks in a room, their voice doesn't magically appear in everyone's ears—it travels through the air, is heard by others in the same room, and is also recorded in the party's official transcript. The chat system operates similarly: a message originates from one user, travels through the server infrastructure, reaches other users in the same room, and is permanently archived for later reference.

The following diagram illustrates this journey:



Step-by-Step Journey of a Chat Message

1. **User Input and Client Preparation**
   - Alice types "Hello everyone!" into her chat interface and presses Enter.
   - The client application (browser) creates a structured `ChatMessage` object containing:
     - `sender`: "alice" (her authenticated username)
     - `content`: "Hello everyone!" (validated for length ≤ `MAX_MESSAGE_LENGTH`)
     - `roomId`: "general" (the current room she's viewing)
     - `id`: 0 (temporary client-side ID, will be replaced by server)
     - `timestamp`: (initially empty, will be set by server)
   - The client wraps this `ChatMessage` in a `WebSocketMessage` with:
     - `type`: `EVENT_CHAT_MESSAGE` ("chat_message")
     - `payload`: The `ChatMessage` object
     - `timestamp`: Current client ISO timestamp (for debugging, not trusted)
2. **WebSocket Transmission**
   - The client's WebSocket connection (already established and authenticated) calls `socket.send()` with the JSON-stringified `WebSocketMessage`.
   - The message travels over the persistent TCP connection established during the initial handshake, avoiding the overhead of HTTP headers and connection setup.
3. **Server Reception and Validation**
   - The WebSocket server receives the raw message string on Alice's specific socket.

- `ConnectionManager.handleMessage(socket, rawMessage)` is invoked:
  - **Parsing**: `parseAndValidateWebSocketMessage(rawMessage)` attempts to parse the JSON. If parsing fails (malformed JSON), the server sends an error response to Alice's connection only and stops processing.
  - **Validation**: The method validates the message has required fields (`type` , `payload`) and that `type` equals `EVENT_CHAT_MESSAGE` .
  - **Authentication Check**: The server looks up Alice's `ClientSession` using the socket connection ID, confirming she's authenticated and currently in the "general" room (matching the `roomId` in the payload).
  - **Payload Validation**: `validateChatMessagePayload(payload)` checks the `ChatMessage` structure: `content` length, `sender` matches authenticated user (preventing impersonation), and `roomId` exists.

4. **Message Processing and Storage**
   - The server calls `createChatMessage(sender, content, roomId)` to create a canonical message:
     - Generates a server-assigned unique ID (sequential or UUID)
     - Sets the authoritative `timestamp` to `new Date().toISOString()` (server time, preventing clock skew issues)
     - Uses the validated `sender` and `content`
   - The Room Manager receives the processed message and calls `storeMessage(message)` :
     - The Persistence Service stores the complete `ChatMessage` (with server-set ID and timestamp) in the database.
     - Storage happens synchronously before broadcasting to ensure no message is lost if the server crashes immediately after broadcasting.
5. **Recipient Determination and Broadcast**
   - `RoomManager.broadcastToRoom(roomId, message, excludeConnectionId)` is called with:
     - `roomId` : "general"
     - `message` : The server-created `ChatMessage`
     - `excludeConnectionId` : Alice's connection ID (so she doesn't receive her own message echo)
   - The Room Manager retrieves the `memberIds` Set for room "general".
   - For each member ID except Alice's:
     - The Room Manager calls `ConnectionManager.sendToClient(connectionId, message)`
     - The Connection Manager looks up the WebSocket for that connection ID
     - It checks the socket's `readyState` (must be `OPEN`). If closed, it silently skips this recipient (user may have disconnected mid-broadcast).
     - It wraps the `ChatMessage` in a server `WebSocketMessage` using `createServerMessage(EVENT_CHAT_MESSAGE, chatMessage)`
6. **Network Delivery to Recipients**
   - For each active recipient (e.g., Bob, Charlie), the server sends the JSON message through their individual WebSocket connections.
   - The messages travel asynchronously and may arrive at slightly different times due to network conditions, but all contain the same server timestamp for consistent ordering.
7. **Client Reception and Rendering**
   - Bob's browser receives the WebSocket message via the `onmessage` event.
   - His client parses the JSON, validates the structure, and extracts the `ChatMessage` .
   - The UI updates to display:
     - Sender name: "alice" (styled according to user preferences)
     - Message content: "Hello everyone!"
     - Timestamp: Formatted locally (e.g., "2:30 PM") from the ISO timestamp
     - Visual indication that this is a new message (highlight, notification sound if room is active)
   - The client may store the message in local memory for the current session but relies on server history for persistence.
8. **Completion and Cleanup**
   - The server logs the message delivery (optional, for debugging).
   - All resources (database connections, memory objects) are released.
   - The system is ready for the next message.

Critical Design Insight:

> The server acts as the **single source of truth** for message identity and timing. By generating the final ID and timestamp server-side, we prevent conflicts that could arise from multiple clients generating IDs independently. This also prevents malicious clients from backdating messages or forging sender identities.

Common Variations and Edge Cases:

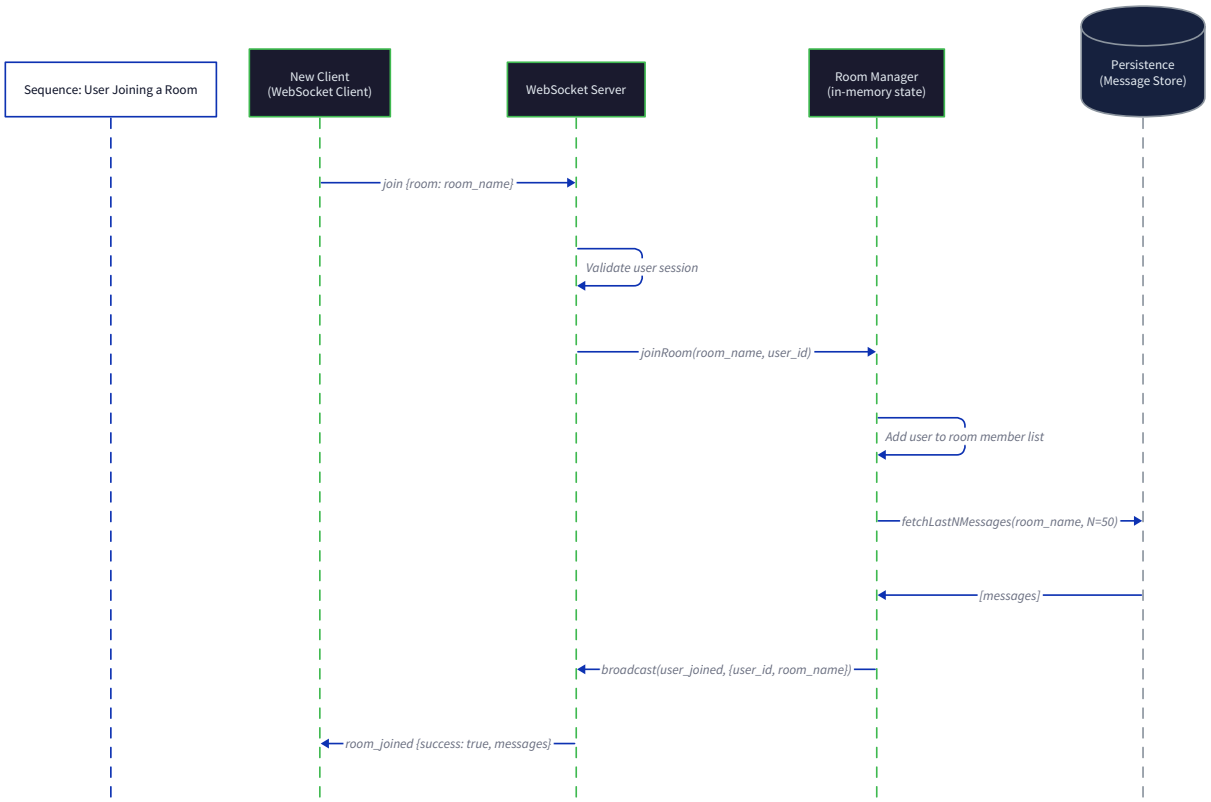
- **Empty Room**: If Alice is alone in the room, `memberIds` contains only her ID, so after excluding her, broadcast sends to zero recipients. The message is still stored for history.

- **Partial Delivery**: If Bob's connection drops during transmission, his socket won't be in `OPEN` state, so he misses the real-time message but can retrieve it later via message history.
- **High Volume**: Under heavy load, the store-broadcast sequence remains atomic per message but multiple messages may be processed concurrently, potentially delivering messages slightly out of order (though timestamps provide correct ordering for display).

Sequence: Joining a Room and Loading History

Mental Model: Entering a Conference Hall Imagine arriving at a large conference center. You first check in at the registration desk (authentication), receive your badge (session), then find the specific breakout room you want to join. As you enter, the ongoing conversation pauses briefly—the moderator announces your arrival, provides you with the last few minutes of discussion notes (message history), and adds you to the attendee list. Other participants now see you as present and can direct messages to you.

The following diagram illustrates this process:



Step-by-Step Room Joining Process

1. **User Interface Action**
   - Bob, already authenticated and connected via WebSocket, clicks "Join General Chat" in the room browser interface.
   - The client application sends a `WebSocketMessage` with:
     - `type`: "join_room"
     - `payload`: `{ roomId: "general" }`
     - (Optional) `timestamp`: Client current time
2. **Server Authentication and Validation**
   - The WebSocket server receives the message and routes it to `ConnectionManager.handleMessage`.
   - The server validates:
     - Bob has an active `ClientSession` (authenticated during connection upgrade)
     - His session isn't expired (checked against `sessionTTL`)
     - He has permission to join the room (in this basic implementation, all public rooms are joinable)
   - The server extracts Bob's current `roomId` from his `ClientSession` (may be `null` if he's in no room, or a previous room if he's switching).

3. **Room Transition Management**
   - If Bob is currently in another room (e.g., "random"):
     - The server calls `RoomManager.leaveRoom(connectionId, previousRoomId)`
     - This removes Bob's `connectionId` from the `memberIds` Set of "random"
     - Broadcasts `EVENT_USER_LEFT` to remaining members of "random"
     - Updates Bob's `ClientSession.roomId` to `null`
   - The server now processes the join request by calling `RoomManager.joinRoom(connectionId, "general", userId)`
4. **Room Membership Update**
   - The Room Manager retrieves or creates room "general":
     - If the room doesn't exist, `createRoom("general")` is called, initializing an empty `memberIds` Set.
   - Bob's `connectionId` is added to the room's `memberIds` Set.
   - Bob's `ClientSession.roomId` is updated to "general".
   - The Room Manager retrieves the list of current member user IDs (excluding Bob) for later notification.
5. **History Retrieval**
   - The Room Manager calls `getRecentMessagesForRoom("general", limit)` where `limit` is configurable (e.g., 50 messages).
   - The Persistence Service queries the database for messages where `roomId = "general"`, ordered by `timestamp` descending, limited to 50.
   - Results are returned as an array of complete `ChatMessage` objects with server-generated IDs and timestamps.
   - The messages are sorted ascending (oldest first) for logical presentation to the user.
6. **Presence Notification to Existing Members**
   - The Room Manager creates a `user_joined` notification containing:
     - `userId`: Bob's username
     - `roomId`: "general"
     - `timestamp`: Current server time
     - (Optional) `userCount`: Updated member count
   - `RoomManager.broadcastToRoom("general", notification, excludeConnectionId)` is called where `excludeConnectionId` is Bob's connection ID (he doesn't need to notify himself).
   - All existing room members receive the `EVENT_USER_JOINED` message and update their UI to show Bob as present (add to member list, possibly show "Bob joined" system message).
7. **Join Success Response to Joining User**
   - The server prepares a comprehensive "room_joined" success response containing:
     - `room`: Room details (name, ID, member count)
     - `members`: Array of current member usernames (for populating the sidebar)
     - `history`: The retrieved message history array
     - `timestamp`: Server time of join completion
   - `ConnectionManager.sendToClient` delivers this specifically to Bob's connection.
8. **Client-Side State Synchronization**
   - Bob's client receives the "room_joined" response and:
     - Updates the current room view to show "general" as active
     - Populates the message history area with the 50 historical messages (oldest at top, newest at bottom)
     - Updates the member sidebar with the list of current members
     - (Optional) Marks any unread indicators for this room as read
   - The client may now enable the message input box (previously disabled if not in a room).
9. **Post-Join Cleanup and Optimization**
   - The server may perform maintenance:
     - Clean up empty rooms if all members have left (configurable policy)
     - Update room activity timestamp for "least recently used" tracking
     - Log the join event for analytics (optional)

Critical Design Insight:

> The join operation is **atomic from the user's perspective** but involves multiple non-atomic steps internally. We carefully sequence operations to ensure Bob receives history before any new messages sent after his join, preventing race conditions where he might miss messages sent during his join process.

Data Flow Table: Room Join Operations

Step	Component Involved	Data Transformed	State Change
1. Client request	Client Application	UI event → `WebSocketMessage`	Client marks room as "joining" (disables UI)
2. Auth validation	Connection Manager	Validates session against `ClientSession` store	None (verification only)
3. Leave previous room	Room Manager	Removes `connectionId` from previous room's `memberIds`	Previous room member count decreases
4. Join new room	Room Manager	Adds `connectionId` to new room's `memberIds`	New room member count increases
5. History retrieval	Persistence Service	Database query → `ChatMessage[]`	None (read-only)
6. Notify existing members	Room Manager → Connection Manager	Creates `user_joined` event, broadcasts	Other clients update their member lists
7. Send success response	Connection Manager → Client	Packages room data, members, history	Client updates full UI state
8. UI update	Client Application	Renders history, updates room context	User can now participate

Common Edge Cases and Handling:

Scenario	System Response	Rationale
Room doesn't exist	Room Manager creates it with Bob as first member	Lazy room creation simplifies UX; rooms exist only when needed
User already in room	Server sends current room state without re-joining	Idempotent operation prevents duplicate notifications
History too large	Pagination: only recent messages sent initially	Prevents overwhelming client with thousands of messages
Network timeout during join	Client may retry with idempotent join request	Join operation designed to be safely repeatable
User banned from room	Server rejects join with error message (future enhancement)	Basic implementation allows all joins; permission system would intercept here

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
Message Validation	Manual field-by-field checks in JavaScript	JSON Schema validation with libraries like `ajv`
History Pagination	Fixed `LIMIT` clause in SQL query	Cursor-based pagination with timestamp offsets
Member List Sync	Send full list on each join/leave	Differential updates using patch operations
Error Recovery	Simple retry with exponential backoff	Circuit breaker pattern with fallback responses

B. Recommended File Structure:

```
real-time-chat/
├── server/
│   ├── index.js                    # Main server entry point
│   ├── connection/
│   │   ├── manager.js              # ConnectionManager class
│   │   ├── lifecycle.js            # Connection event handlers
│   │   └── validation.js           # parseAndValidateWebSocketMessage, etc.
│   ├── rooms/
│   │   ├── manager.js              # RoomManager class (implements broadcast, join, leave)
│   │   ├── presence.js             # Typing indicators, online status
│   │   └── history.js              # Message history retrieval logic
│   ├── auth/
│   │   ├── service.js              # Authentication service (login, register)
│   │   ├── sessions.js             # Session management (create, validate, invalidate)
│   │   └── middleware.js           # WebSocket upgrade authentication
│   └── persistence/
│       ├── messageStore.js         # storeMessage, getMessagesForRoom
│       ├── userStore.js            # User CRUD operations
│       └── database.js             # Database connection setup
├── client/
│   ├── public/
│   │   ├── index.html              # Main HTML file
│   │   ├── app.js                  # Main client application logic
│   │   ├── ui.js                   # DOM manipulation and rendering
│   │   └── ws-client.js            # WebSocket connection management
│   └── package.json                # Client dependencies (if using build tools)
└── package.json                    # Server dependencies
```

C. Infrastructure Starter Code (Complete Validation Utilities):

JAVASCRIPT

```javascript
// server/connection/validation.js

/**
 * Parses and validates a raw WebSocket message string
 * @param {string} rawMessage - Raw string received from WebSocket
 * @returns {WebSocketMessage} Parsed and validated message object
 * @throws {Error} If message is invalid JSON or missing required fields
 */
function parseAndValidateWebSocketMessage(rawMessage) {
  if (typeof rawMessage !== 'string') {
    throw new Error('Message must be a string');
  }

  let parsed;
  try {
    parsed = JSON.parse(rawMessage);
  } catch (err) {
    throw new Error(`Invalid JSON: ${err.message}`);
  }

  if (!parsed.type || typeof parsed.type !== 'string') {
    throw new Error('Message must have a string "type" field');
  }

  if (parsed.payload === undefined) {
    throw new Error('Message must have a "payload" field');
  }

  // Optional timestamp validation
  if (parsed.timestamp && !isValidISODate(parsed.timestamp)) {
    throw new Error('Invalid timestamp format, expected ISO string');
  }

  return {
    type: parsed.type,
    payload: parsed.payload,
    timestamp: parsed.timestamp || new Date().toISOString()
  };
}

/**
 * Validates a ChatMessage payload structure
 * @param {any} payload - The payload to validate
 * @returns {ChatMessage} Validated chat message
```

```
 * @throws {Error} If payload doesn't match ChatMessage structure
 */
function validateChatMessagePayload(payload) {
  if (!payload || typeof payload !== 'object') {
    throw new Error('Payload must be an object');
  }

  if (!payload.sender || typeof payload.sender !== 'string') {
    throw new Error('Payload must have a string "sender" field');
  }

  if (!payload.content || typeof payload.content !== 'string') {
    throw new Error('Payload must have a string "content" field');
  }

  if (!payload.roomId || typeof payload.roomId !== 'string') {
    throw new Error('Payload must have a string "roomId" field');
  }

  if (payload.content.length > MAX_MESSAGE_LENGTH) {
    throw new Error(`Message content exceeds maximum length of ${MAX_MESSAGE_LENGTH} characters`);
  }

  // Allow client to send ID and timestamp, but they will be overwritten by server
  return {
    sender: payload.sender,
    content: payload.content.trim(), // Trim whitespace
    roomId: payload.roomId,
    id: payload.id || 0, // Temporary, will be replaced
    timestamp: payload.timestamp || null // Will be replaced by server
  };
}

/**
 * Creates a properly formatted server-originated WebSocket message
 * @param {string} type - Message type (use EVENT_* constants)
 * @param {any} payload - The payload to send
 * @returns {WebSocketMessage} Formatted message ready for JSON.stringify
 */
function createServerMessage(type, payload) {
  return {
    type,
    payload,
```

```
    timestamp: new Date().toISOString()
  };
}

/**
 * Creates a new ChatMessage with server-generated ID and timestamp
 * @param {string} sender - Username of sender
 * @param {string} content - Message content (already validated)
 * @param {string} roomId - Room identifier
 * @returns {ChatMessage} Complete chat message object
 */
function createChatMessage(sender, content, roomId) {
  return {
    id: generateMessageId(), // Implement this (could be sequential or UUID)
    sender,
    content: content.trim(),
    roomId,
    timestamp: new Date().toISOString()
  };
}

// Helper function for ID generation (simple sequential for learning)
let messageIdCounter = 1;
function generateMessageId() {
  return messageIdCounter++;
}

// Helper function for ISO date validation
function isValidISODate(dateString) {
  return !isNaN(Date.parse(dateString));
}

module.exports = {
  parseAndValidateWebSocketMessage,
  validateChatMessagePayload,
  createServerMessage,
  createChatMessage,
  MAX_MESSAGE_LENGTH: 1000
};
```

D. Core Logic Skeleton Code:

JAVASCRIPT

```javascript
// server/rooms/manager.js

class RoomManager {
  constructor(messageStore, connectionManager) {
    this.rooms = new Map(); // roomId -> Room object
    this.messageStore = messageStore;
    this.connectionManager = connectionManager;
  }

  /**
   * Adds a user to a room, fetches history, and notifies other members
   * @param {string} connectionId - The WebSocket connection ID
   * @param {string} roomId - Room identifier to join
   * @param {string} userId - Authenticated user ID
   * @returns {Promise<Room>} The joined room object
   */
  async joinRoom(connectionId, roomId, userId) {
    // TODO 1: Validate parameters (non-empty strings)
    // TODO 2: Get or create the room (check this.rooms Map, create if missing using createRoom)
    // TODO 3: Check if user is already in room (optional optimization)
    // TODO 4: Add connectionId to room.memberIds Set
    // TODO 5: Fetch recent message history (call this.messageStore.getRecentMessagesForRoom)
    // TODO 6: Get list of current member user IDs (excluding the joining user)
    // TODO 7: Create user_joined notification using createServerMessage(EVENT_USER_JOINED, ...)
    // TODO 8: Broadcast notification to all other room members using this.broadcastToRoom
    // TODO 9: Prepare success response with room info, members list, and history
    // TODO 10: Return room object for further processing
  }

  /**
   * Sends a message to all members of a room, optionally excluding sender
   * @param {string} roomId - Target room identifier
   * @param {ChatMessage} message - The chat message to broadcast
   * @param {string} [excludeConnectionId] - Optional connection ID to exclude (usually sender)
   * @returns {void}
   */
  broadcastToRoom(roomId, message, excludeConnectionId) {
    // TODO 1: Validate roomId exists in this.rooms Map
    // TODO 2: Get the room object and its memberIds Set
    // TODO 3: Convert memberIds Set to array for iteration
    // TODO 4: For each connectionId in memberIds:
    //   a. Skip if connectionId === excludeConnectionId
```

```
    //   b. Get WebSocket via this.connectionManager.getSocket(connectionId)
    //   c. Check socket is open (readyState === WebSocket.OPEN)
    //   d. Create formatted message using createServerMessage(EVENT_CHAT_MESSAGE, message)
    //   e. Send via this.connectionManager.sendToClient(connectionId, formattedMessage)
    // TODO 5: Log broadcast statistics (optional)
  }

  /**
   * Removes a user from a room and notifies remaining members
   * @param {string} connectionId - The WebSocket connection ID to remove
   * @param {string} roomId - Room identifier
   * @returns {void}
   */
  leaveRoom(connectionId, roomId) {
    // TODO 1: Validate room exists in this.rooms Map
    // TODO 2: Remove connectionId from room.memberIds Set
    // TODO 3: If room is now empty, consider cleanup (optional: delete room or mark inactive)
    // TODO 4: Create user_left notification using createServerMessage(EVENT_USER_LEFT, ...)
    // TODO 5: Broadcast notification to remaining room members
    // TODO 6: Update connection's session to reflect leaving room (via connectionManager)
  }

  /**
   * Updates and broadcasts a user's typing status in a room
   * @param {string} connectionId - The WebSocket connection ID
   * @param {string} roomId - Room identifier
   * @param {boolean} isTyping - Whether user is currently typing
   * @returns {void}
   */
  setUserTyping(connectionId, roomId, isTyping) {
    // TODO 1: Validate user is actually in the specified room
    // TODO 2: Create typing notification with user info and isTyping flag
    // TODO 3: Broadcast to room members excluding the typing user
    // TODO 4: (Advanced) Implement debouncing to prevent spam - only send if state changed
    // TODO 5: (Advanced) Set timeout to automatically send "stopped typing" after inactivity
  }
}

module.exports = RoomManager;
```

JAVASCRIPT

```javascript
// server/connection/manager.js

class ConnectionManager {
  constructor() {
    this.connections = new Map(); // connectionId -> ClientSession
  }

  /**
   * Main entry point for handling incoming WebSocket messages
   * @param {string} connectionId - The connection ID (from socket)
   * @param {string} rawMessage - Raw WebSocket message string
   * @returns {void}
   */
  handleMessage(connectionId, rawMessage) {
    // TODO 1: Look up ClientSession from this.connections using connectionId
    // TODO 2: Parse and validate raw message using parseAndValidateWebSocketMessage
    // TODO 3: Validate user session is still active (check lastActivity against sessionTTL)
    // TODO 4: Route message based on type:
    //   - If type === 'chat_message': validate payload, create chat message, store, broadcast
    //   - If type === 'join_room': extract roomId, call roomManager.joinRoom
    //   - If type === 'typing': extract roomId and isTyping, call roomManager.setUserTyping
    //   - Unknown type: send error response
    // TODO 5: Update lastActivity timestamp in ClientSession
    // TODO 6: Handle any errors: send appropriate error message to client, log error
  }

  /**
   * Sends a message to a specific client if their socket is open
   * @param {string} connectionId - Target connection ID
   * @param {WebSocketMessage} message - Message to send
   * @returns {boolean} True if message was sent, false if socket not open
   */
  sendToClient(connectionId, message) {
    // TODO 1: Look up ClientSession from this.connections
    // TODO 2: Get WebSocket from ClientSession.socket
    // TODO 3: Check socket.readyState === WebSocket.OPEN
    // TODO 4: If open: socket.send(JSON.stringify(message)), return true
    // TODO 5: If not open: return false (connection will be cleaned up by heartbeat check)
    // TODO 6: Wrap in try-catch for network errors
  }
}
```

```
module.exports = ConnectionManager;
```

E. Language-Specific Hints (JavaScript/Node.js):

1. **WebSocket State Checking**: Always check `socket.readyState === WebSocket.OPEN` before sending. The WebSocket protocol has states: CONNECTING (0), OPEN (1), CLOSING (2), CLOSED (3).
2. **Error Handling in Async Flow**: Use try-catch blocks around database operations and JSON parsing. Send descriptive error messages to clients for debugging but log full errors server-side.
3. **Set Operations for Membership**: JavaScript's `Set` is ideal for `memberIds` because it ensures uniqueness and has O(1) add/delete/has operations. Convert to array only when needed for iteration.
4. **Timestamp Consistency**: Use `new Date().toISOString()` for all server-generated timestamps. This produces UTC time in a standard format that's easily sortable and timezone-agnostic.
5. **Connection Cleanup**: Implement periodic cleanup (every 30 seconds) to remove stale connections by checking `ClientSession.lastActivity` against current time.

F. Milestone Checkpoint for Message Flow:

After implementing Milestone 2 (Message Broadcasting):

1. **Start the server**: `node server/index.js`
2. **Open two browser tabs** to `http://localhost:3000`
3. **Log in with different usernames** in each tab
4. **Send a message from Tab 1** and verify:
   - Message appears in Tab 1's chat window (echo from server optional)
   - Message appears in Tab 2's chat window within ~100ms
   - Message shows correct sender name and timestamp
   - Console logs show no validation errors
5. **Test edge cases**:
   - Send empty message → should be rejected
   - Send message > 1000 characters → should be rejected
   - Disconnect Tab 2 network → message should fail to deliver but not crash server
   - Reconnect Tab 2 → should see message in history when rejoining room

G. Debugging Tips for Data Flow Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Messages appear only for sender	`broadcastToRoom` is excluding everyone	Log `memberIds` size before and after exclusion	Check `excludeConnectionId` logic; verify room membership
History loads but real-time messages don't appear	Client not handling `EVENT_CHAT_MESSAGE` events	Check client WebSocket `onmessage` handler	Add handler for `type === 'chat_message'`
User count wrong after join/leave	Race condition in member tracking	Add debug logs to `joinRoom` and `leaveRoom`	Ensure atomic operations or add mutex for room updates
Messages appear out of order	Multiple messages processed concurrently	Log server timestamps vs arrival order	Sort by timestamp on client side before display
Join takes long time	History query is slow with many messages	Profile database query time	Add index on `(roomId, timestamp)` columns

Error Handling and Edge Cases

> **Milestone(s):** All milestones (1-4). Robust error handling and edge case management are critical for the stability and reliability of the real-time chat system, ensuring graceful degradation when things go wrong rather than catastrophic failure.

Imagine a busy office building where the telephone system suddenly fails—calls drop mid-conversation, some lines get crossed, and occasionally wrong numbers slip through. A well-designed system has contingency plans: operators who can reconnect calls, verification procedures to prevent wrong connections, and capacity limits to avoid overload. Similarly, our chat application must anticipate and handle failures gracefully because real-time systems operate in an inherently unreliable

environment—networks fail, servers crash, users send invalid data, and edge cases emerge under load. This section details how the system detects, responds to, and recovers from expected failures and unexpected conditions.

Common Failure Modes and Recovery

The chat system faces several predictable failure modes. Proactively designing for these scenarios prevents cascading failures and maintains user experience. Our approach follows the principle of **defensive programming**: validate everything, assume nothing, and always have a recovery path.

Network Drops and Unclean Disconnections

Mental Model: The Frayed Telephone Cord
Think of WebSocket connections as physical telephone cords that can be accidentally unplugged or fray over time. The server needs to detect when a cord is cut (connection lost) and clean up the call (user session) to avoid "ghost users" lingering in rooms.

Network failures are the most common issue in distributed systems. Clients may lose connectivity due to Wi-Fi drops, mobile network switching, or firewall timeouts without sending a proper WebSocket close frame.

Failure Mode	Detection Method	Recovery Strategy	Rationale
Silent Connection Death (client disappears without `onclose`)	Heartbeat (ping/pong) mechanism: Server sends periodic ping; if no pong received within timeout, marks connection dead	Server proactively cleans up connection state via `ConnectionManager.removeConnection()`; broadcasts `EVENT_USER_LEFT` to affected rooms	Prevents memory leaks and ensures presence accuracy without relying on TCP timeouts (which can be minutes)
Intermittent Network Flakiness (brief packet loss)	Multiple missed heartbeats or repeated WebSocket frame errors	Client implements exponential backoff reconnection logic; server maintains session briefly during short outages (30-60 seconds)	Balances quick recovery with avoiding churn for temporary issues; exponential backoff prevents overwhelming server during network restoration
Asymmetric Connectivity (client can receive but not send)	Client-side heartbeat: client sends periodic "alive" messages; server notices missing messages	Server eventually treats as dead connection after timeout; client detects send failures and initiates reconnection	Detects one-way failures that server ping might miss (since pings require round-trip)

Architecture Decision Record: Implementing Heartbeat vs. Relying on TCP Keepalive

> **Decision: Application-Level Heartbeat Over TCP Keepalive**
>
> - **Context**: WebSocket connections can appear open at TCP level while the application is stuck (browser tab frozen, mobile app backgrounded). We need to detect actual application liveliness, not just TCP connectivity.
> - **Options Considered**:
>   1. **Rely solely on TCP keepalive**: Use operating system TCP keepalive packets (default 2 hours idle timeout).
>   2. **WebSocket protocol ping/pong**: Use built-in WebSocket control frames (RFC 6455 Section 5.5.2).
>   3. **Application-level heartbeat messages**: Custom `{type: "heartbeat"}` messages in the chat protocol.
> - **Decision**: Use WebSocket protocol ping/pong frames for connection health, supplemented with application-level activity tracking.
> - **Rationale**: WebSocket ping/pong is standardized, doesn't clutter application message protocol, and works across all WebSocket implementations. Application-level activity tracking (last message timestamp) helps distinguish idle from dead connections. TCP keepalive is too slow (hours vs seconds).
> - **Consequences**: Requires WebSocket library that exposes ping/pong API; adds periodic server workload; ensures connections are cleaned within 30-60 seconds of failure.

Option	Pros	Cons	Chosen?
TCP Keepalive	No application code, operating system handles it	Extremely slow detection (default 2+ hours), not configurable per-connection	❌
WebSocket Ping/Pong	Standardized, efficient (control frames), most libraries support	Some browser WebSocket APIs don't expose ping/pong to JavaScript	✅ Primary
Application Heartbeat	Full control, works everywhere, can carry metadata	Adds to message count, requires protocol design, redundant with ping/pong	✅ Supplementary

Implementation Strategy:

1. Server sends ping every 25 seconds to each connection
2. If no pong received within 30 seconds, mark connection as stale

3. After 60 seconds total (two missed cycles), forcibly close connection and clean up
4. Track `lastActivity` timestamp in `ClientSession` updated on any message receive
5. Background cleaner runs every minute to remove sessions inactive > 60 seconds

Server Crashes and Restarts

Mental Model: The Office Power Outage
When the power goes out in an office building, everything stops instantly. When power returns, systems need to reboot, restore essential services, and notify people that there was an interruption.

Server crashes (unhandled exceptions, OOM killer, deployment restarts) terminate all in-memory state. The recovery focus is on data persistence and graceful client reconnection.

Failure Mode	Detection Method	Recovery Strategy	Rationale
Process Crash (Node.js uncaught exception)	Process exits; process manager (PM2, systemd) detects exit	Process manager restarts server; server reloads persistent data (rooms, messages); clients automatically reconnect	Minimizes downtime; stateless WebSocket connections require full client reconnection
Memory Exhaustion (OOM)	Monitoring memory usage approaching limits	Implement connection limits (`maxConnections`), message size limits, and aggressive cleanup of idle connections	Prevents catastrophic failure; allows server to reject new connections rather than crash
Dependency Failure (database connection lost)	Database client library emits error events	Continue serving existing connections with degraded functionality (no history queries); queue or drop persistence operations	Graceful degradation maintains real-time chat while persistence is temporarily unavailable

Recovery Procedure on Server Startup:

1. Load recent messages from database for active rooms (cached in memory)
2. Recreate room structures from persisted room definitions (if stored)
3. Initialize empty connection maps (all clients must reconnect)
4. Begin accepting new WebSocket connections
5. Optional: Broadcast "server was restarted" to reconnected clients

> **Key Insight**: The system is designed to be **mostly stateless at the connection layer**. All essential state (messages, user accounts, room definitions) persists to storage, while ephemeral state (who's currently connected, typing status) rebuilds as clients reconnect. This aligns with the **Twelve-Factor App** principle of disposable processes.

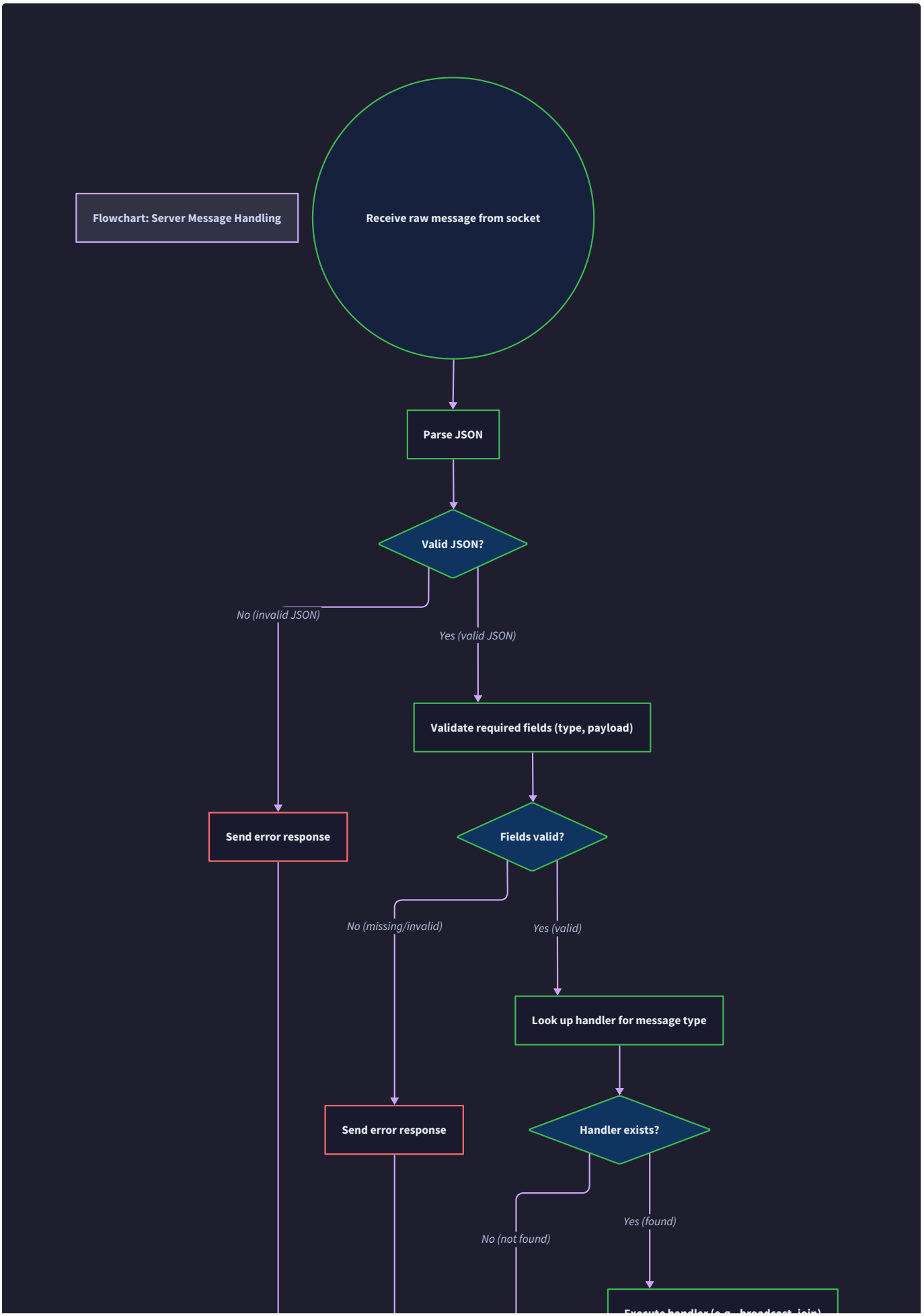
Malformed and Malicious Messages

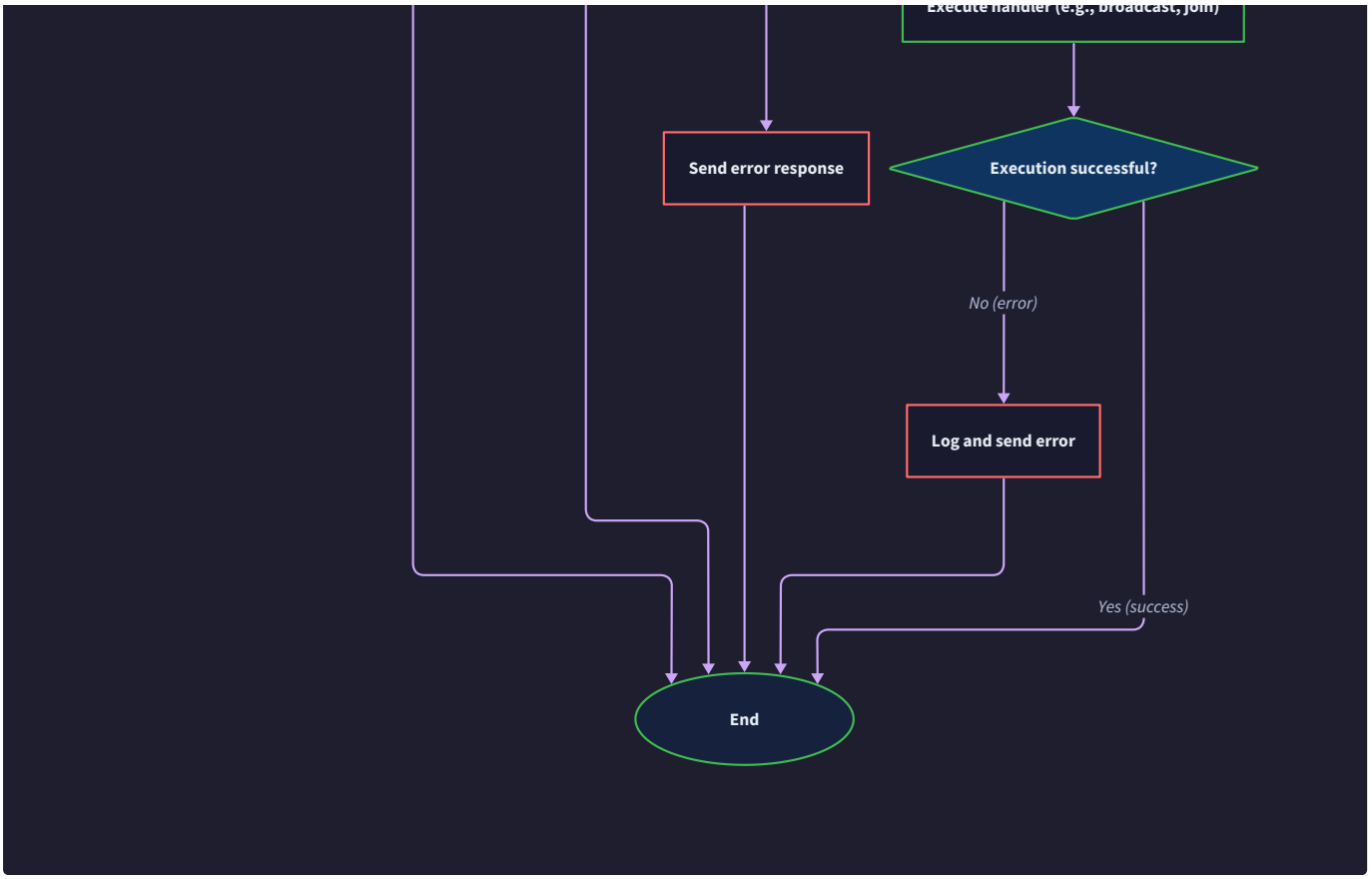
Mental Model: The Mailroom Screening Process
Every incoming package passes through security screening: checking for hazardous materials, verifying sender identity, and ensuring proper addressing. Similarly, every WebSocket message undergoes validation before processing.

Malformed messages (invalid JSON, missing fields) and malicious payloads (injection attempts, enormous strings) can crash the server or exploit vulnerabilities if not properly handled.

Failure Mode	Detection Method	Recovery Strategy	Rationale
Invalid JSON Syntax	`JSON.parse()` throws SyntaxError in `parseAndValidateWebSocketMessage()`	Catch exception, send error response `{type: "error", payload: {code: "INVALID_JSON"}}` to client, keep connection open	Prevents server crash; informs client of protocol violation without disconnecting
Schema Violation (missing required fields, wrong types)	`validateChatMessagePayload()` checks structure against expected schema	Reject message with descriptive error, log for monitoring, continue processing other messages	Maintains protocol consistency; helps developers debug client implementation
Message Too Large (exceeds `MAX_MESSAGE_LENGTH`)	Check `payload.content.length` before validation	Immediately reject without full parsing, send error, optionally disconnect abusive clients	Protects against memory exhaustion attacks and network congestion
Injection Attempts (JavaScript, HTML in messages)	Client-side sanitization preferred; server validation for critical fields	Store raw content but escape before sending to other clients; log suspicious patterns	Allows flexibility in client rendering while preventing XSS in web clients

Message Validation Pipeline (referencing


Flowchart: Server Message Handling
Receive raw message from socket
Parse JSON
Valid JSON?
No (invalid JSON)
Yes (valid JSON)
Validate required fields (type, payload)
Send error response
Fields valid?
No (missing/invalid)
Yes (valid)
Look up handler for message type
Send error response
Handler exists?
No (not found)
Yes (found)



):

1. **Raw Message Reception**: `ConnectionManager.handleMessage()` receives string
2. **Syntax Validation**: `parseAndValidateWebSocketMessage()` attempts JSON parsing
   - Failure → Send error response, stop processing
3. **Structure Validation**: Verify `WebSocketMessage` has `type` and `payload` fields
   - Missing → Send error, stop
4. **Type-Specific Validation**: Route to appropriate validator based on `type`
   - `EVENT_CHAT_MESSAGE` → `validateChatMessagePayload()` checks sender, content length, roomId
   - `EVENT_USER_TYPING` → Validate boolean flag and room membership
5. **Business Logic Validation**: Check application state (is user in room? authenticated?)
6. **Processing**: Only after all validation passes, execute handler

Error Response Format:

```javascript
// Server sends this when message validation fails
{
  type: "error",
  payload: {
    code: "VALIDATION_ERROR", // or "NOT_IN_ROOM", "UNAUTHENTICATED"
    message: "Content exceeds maximum length of 1000 characters",
    originalType: "chat_message", // The failed message type
    timestamp: "2023-10-05T14:30:00Z"
  }
}
```

Resource Exhaustion (Memory, CPU, File Descriptors)

Mental Model: The Concert Venue Capacity Manager

A venue has maximum capacity for safety. When full, new attendees wait in line. Staff monitor crowd density and may ask overly rowdy patrons to leave. Similarly, the server must enforce limits to remain stable.

Node.js has single-threaded event loop and memory constraints. Resource exhaustion can degrade performance for all users or cause complete failure.

Resource	Monitoring	Protection Strategy	Mitigation
Memory	`process.memoryUsage()`, monitor RSS	Limit concurrent connections; implement message size limits; use efficient data structures (`Map` over `Object`, `Set` for member lists)	Connection rejection when near limit; aggressive garbage collection by clearing message history caches
CPU	Event loop lag measurement; profiling	Rate limiting per connection (messages/second); offload expensive operations (password hashing) to worker threads	Shed load by rejecting new connections; use efficient algorithms (O(1) lookups in connection maps)
File Descriptors	`ulimit` monitoring; connection count tracking	Set `maxConnections` below system limit (typically 65% of `ulimit -n`)	Graceful degradation: "Server busy" response during HTTP upgrade
Database Connections	Connection pool metrics	Implement connection pooling with reasonable max; queue operations when pool exhausted	Return partial functionality (real-time works, history may fail)

Architecture Decision Record: In-Memory State vs. External Storage for Active Connections

> **Decision: In-Memory Connection State with Single-Server Simplicity**
>
> - **Context**: The learning-focused chat application runs as a single Node.js process. We need to track all active WebSocket connections, their room memberships, and typing status.
> - **Options Considered**:
>   1. **Pure in-memory**: Store everything in process memory using `Map` and `Set` collections.
>   2. **Shared Redis**: Store connection state in Redis for potential multi-server scaling.
>   3. **Hybrid approach**: Critical state (room membership) in Redis, ephemeral state (typing) in memory.
> - **Decision**: Pure in-memory state for all connection data.
> - **Rationale**: For the learning context and single-server deployment, in-memory is simpler with zero external dependencies, faster (no network latency), and sufficient for hundreds to thousands of concurrent connections. The complexity of distributed state management outweighs benefits at this scale.
> - **Consequences**: Server restart loses all connection state (users must reconnect); vertical scaling only (single server); cannot scale horizontally without redesign.

Option	Pros	Cons	Chosen?
Pure In-Memory	Simplest, fastest, no external dependencies	Lost on server restart, doesn't scale horizontally	✅ For learning context
Shared Redis	Survives server restarts, enables horizontal scaling	Additional infrastructure, network latency, complexity	❌ Over-engineering
Hybrid	Balances performance and durability	Increased complexity, consistency challenges	❌ Intermediate complexity not justified

Edge Case Analysis

Edge cases are unusual but valid scenarios that test the boundaries of the system design. Proper handling distinguishes a robust system from a fragile one.

Duplicate Usernames and Connection Conflicts

Scenario: Two users register with the same username "Alice", or the same user logs in from two devices (browser and phone).

Edge Case	Problem	Solution
Registration Duplicate	Two users with identical `username` in database	Unique constraint on `username` field in database; `registerUser()` returns error "Username taken"
Simultaneous Connections Same user from multiple devices	Both connections receive messages; typing indicators conflict; presence shows as single user	Treat each connection independently: separate `ClientSession` objects, both receive broadcasts, typing status from either device shows as "Alice is typing"
Same Username in Same Room	Confusion about which user sent a message	Use unique internal `userId` (database ID) for all routing; display username but include `userId` in message metadata for disambiguation
Username Change While Connected	Other users see old username in cached messages	Store `userId` in `ChatMessage`, not username; fetch current username from database when displaying historical messages

Handling Algorithm for Multiple Connections:

1. When user authenticates, create new `ClientSession` with unique `connectionId`
2. Both sessions have same `userId` but different `connectionId`
3. `RoomManager.broadcastToRoom()` sends to all connections in room
4. Typing indicator: `setUserTyping()` updates status per `connectionId`, broadcasts to room
5. User leaves: only remove specific `connectionId` from room; user remains in room if other connections exist
6. All connections close: `leaveAllRooms()` called for each, user fully removed

Message Size and Rate Extremes

Scenario: A user pastes a 10MB text file, or a bot sends 1000 messages per second.

Edge Case	Problem	Solution
Extremely Long Message (> `MAX_MESSAGE_LENGTH`)	Network congestion, database bloat, UI rendering issues	Server enforces `MAX_MESSAGE_LENGTH` (1000 chars) in `validateChatMessagePayload()`; client also validates before sending
Rapid Message Flood from single connection	Event loop blocking, other users overwhelmed with notifications	Rate limiting: track messages per second per connection; delay or drop excess messages with error "Rate limit exceeded"
Empty Messages (zero-length or whitespace-only)	Spam, confusing UI display	Validation requires `content.trim().length > 0`; reject with descriptive error
Special Character Bomb (emojis, Unicode)	Encoding issues, storage size mismatch (UTF-8 vs byte length)	Count Unicode code points, not bytes, for length limit; normalize encoding before storage

Rate Limiting Implementation:

```javascript
// In ConnectionManager
const messageCounts = new Map(); // connectionId -> {count, resetTime}

function checkRateLimit(connectionId) {
  const now = Date.now();
  let record = messageCounts.get(connectionId);

  if (!record || now > record.resetTime + 60000) {
    // New minute window
    record = { count: 0, resetTime: now };
    messageCounts.set(connectionId, record);
  }

  record.count++;
  return record.count <= 100; // 100 messages per minute limit
}
```

JAVASCRIPT

Rapid Join/Leave Churn

Scenario: A user rapidly joins and leaves a room, or a script creates/destroys rooms quickly.

Edge Case	Problem	Solution
Join-Leave Loop (user spamming enter/exit)	Notification storms for other users; unnecessary database queries for history	Debounce join operations: ignore repeated `joinRoom` calls within 1 second for same user+room
Room Creation Flood	Exhaustion of resources, polluted room lists	Limit room creation rate per user; require unique room names; auto-cleanup empty rooms after timeout
Concurrent Join from Multiple Devices	Race conditions in member list updates	Use atomic operations: `Room.memberIds` as `Set` with `add()` / `delete()`; broadcast only after successful addition
Leave During Join (user leaves while history is loading)	Orphaned operations, inconsistent state	Cancel pending history fetch if connection closes; verify user still in room before broadcasting join notification

Debouncing Algorithm for Room Operations:

```javascript
// In RoomManager
const lastJoinTime = new Map(); // userId+roomId -> timestamp

async function joinRoom(connectionId, roomId, userId) {
  const key = `${userId}:${roomId}`;
  const now = Date.now();
  const lastTime = lastJoinTime.get(key) || 0;

  if (now - lastTime < 1000) { // 1 second debounce
    throw new Error('Please wait before rejoining');
  }

  lastJoinTime.set(key, now);
  // ... rest of join logic
}
```

Clock Skew and Message Ordering

Scenario: Client and server clocks differ by minutes, or messages arrive out of order due to network latency.

Edge Case	Problem	Solution
Client Clock Wrong (behind/ahead)	Message timestamps inconsistent across users	Always use server-assigned timestamp (`Date.now()` on server) in `createChatMessage()` ; ignore client-provided timestamps
Network Reordering (later message arrives first)	Chat history shows messages out of logical order	Use monotonic increasing message IDs (database sequence) for ordering; timestamp for display only
Daylight Saving Time Changes	Displayed times jump forward/backward	Store all timestamps in UTC ISO format; convert to local time only at display layer (client-side)
High-Precision Timing (multiple messages in same millisecond)	Indistinguishable ordering at millisecond level	Use composite key: `timestamp + sequence` or database auto-increment ID

> **Key Insight**: **Server-generated timestamps** are non-negotiable for consistency. The server acts as the "single source of truth" for temporal ordering. Clients may display relative times ("2 minutes ago") to mask small synchronization issues.

Empty Rooms and Zombie Connections

Scenario: All users leave a room but it persists in memory, or connections remain in lists after network failure.

Edge Case	Problem	Solution
Empty Room Accumulation	Memory leak, polluted room listing	Background job removes rooms with `memberIds.size === 0` after 24 hours (or shorter for temporary rooms)
Zombie Connections in member lists	Users appear present but cannot receive messages	`RoomManager` validates connection existence before broadcasting; periodic cleanup removes dead connections from all rooms
Room Name Collisions (case sensitivity)	"General" vs "general" treated as different rooms	Normalize room names: lowercase, trim before storage; case-insensitive matching for joins
Special Character Room Names	Injection, URL encoding issues	Sanitize room names: allow alphanumeric, hyphens, underscores; reject special characters

Room Cleanup Algorithm:

1. Every hour, scan all rooms in `RoomManager`
2. For rooms with `memberIds.size === 0` :
   - If room created > 24 hours ago, delete it

- If temporary room (prefixed with "temp-"), delete immediately

3. For rooms with members, verify each `connectionId` still exists in `ConnectionManager`
4. Remove invalid connections from room membership

Authentication and Session Edge Cases

Scenario: User logs out on one device while active on another, or session expires mid-conversation.

Edge Case	Problem	Solution
Session Expiry During Connection	User suddenly unauthenticated; messages fail	Periodic session validation in heartbeat; on expiration, send `{type: "session_expired"}` and close WebSocket
Concurrent Logout from Another Device	Race condition: which session wins?	Session invalidation removes all sessions for user; all connections receive logout notification
Reused Session Token (theft/replay)	Unauthorized access	Store sessions in database with expiry; one-time use tokens or token rotation on sensitive operations
Password Change with Active Sessions	Old sessions remain valid (security risk)	Invalidate all sessions on password change; require re-authentication on all devices

Session Validation in Message Flow:

1. Each incoming message includes session token (in WebSocket upgrade or first message)
2. `ConnectionManager` validates token via `validateSession()` before routing
3. If invalid/expired: send error, close connection with code 4001 (custom: "Session expired")
4. Background job `cleanupExpiredSessions()` runs hourly to purge old sessions

Implementation Guidance

Technology Recommendations for Error Handling

Component	Simple Option (Learning)	Production-Ready Option
Connection Heartbeat	Manual ping/pong using `setInterval` and WebSocket `.ping()`	`ws` library built-in heartbeat with configurable intervals
Rate Limiting	In-memory counter with fixed window	Redis-based sliding window with `redis-cell` or token bucket algorithm
Validation	Manual `if` statements checking each field	JSON Schema validation with `ajv` or `zod` for TypeScript
Error Tracking	Console logging	Structured logging with `winston` / `pino` + error aggregation (Sentry)
Process Management	Manual `node server.js`	Process manager (PM2, systemd) with auto-restart and monitoring

Recommended Error Handling Module Structure

```
project-root/
  src/
    server/
      connection-manager.js       ← Heartbeat, rate limiting, validation
      room-manager.js             ← Room cleanup, debouncing logic
      auth-service.js             ← Session validation, cleanup
      persistence/                ← Database error handling
        message-store.js
        session-store.js
      middleware/                 ← Validation middleware
        validate-message.js
        rate-limiter.js
      errors/                     ← Custom error classes
        chat-errors.js
        validation-errors.js
      utils/
        logger.js                 ← Structured logging
```

Core Error Handling Skeleton Code

Complete Starter: Custom Error Classes

JAVASCRIPT

```javascript
// src/errors/chat-errors.js

/**
 * Base class for all chat application errors
 */
class ChatError extends Error {
  constructor(message, code, statusCode = 400) {
    super(message);
    this.name = this.constructor.name;
    this.code = code; // Machine-readable error code
    this.statusCode = statusCode; // HTTP-like status for WebSocket close
    Error.captureStackTrace(this, this.constructor);
  }
}

/**
 * Thrown when message validation fails
 */
class ValidationError extends ChatError {
  constructor(message, field) {
    super(message, 'VALIDATION_ERROR', 400);
    this.field = field; // Which field failed validation
  }
}

/**
 * Thrown when user is not authenticated
 */
class AuthenticationError extends ChatError {
  constructor(message = 'Authentication required') {
    super(message, 'UNAUTHENTICATED', 401);
  }
}

/**
 * Thrown when user lacks permission for an operation
 */
class AuthorizationError extends ChatError {
  constructor(message = 'Permission denied') {
    super(message, 'FORBIDDEN', 403);
  }
}
```

```
/**
 * Thrown when resource not found (room, user, message)
 */
class NotFoundError extends ChatError {
  constructor(resourceType, id) {
    super(`${resourceType} "${id}" not found`, 'NOT_FOUND', 404);
    this.resourceType = resourceType;
    this.resourceId = id;
  }
}

/**
 * Thrown when rate limit exceeded
 */
class RateLimitError extends ChatError {
  constructor(retryAfterSeconds) {
    super('Rate limit exceeded', 'RATE_LIMITED', 429);
    this.retryAfter = retryAfterSeconds;
  }
}

module.exports = {
  ChatError,
  ValidationError,
  AuthenticationError,
  AuthorizationError,
  NotFoundError,
  RateLimitError
};
```

Complete Starter: Structured Logger

JAVASCRIPT

```javascript
// src/utils/logger.js

const winston = require('winston');

const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  transports: [
    new winston.transports.Console({
      format: winston.format.combine(
        winston.format.colorize(),
        winston.format.simple()
      )
    }),
    new winston.transports.File({
      filename: 'logs/error.log',
      level: 'error',
      maxsize: 5242880, // 5MB
      maxFiles: 5
    }),
    new winston.transports.File({
      filename: 'logs/combined.log',
      maxsize: 5242880,
      maxFiles: 5
    })
  ]
});

// Helper for logging WebSocket events
logger.connection = (connectionId, event, details = {}) => {
  logger.info({
    type: 'connection',
    connectionId,
    event,
    ...details
  });
};
```

```
// Helper for logging chat events
logger.chat = (roomId, event, details = {}) => {
  logger.info({
    type: 'chat',
    roomId,
    event,
    ...details
  });
};

module.exports = logger;
```

Core Logic Skeleton: Enhanced Message Validation

JAVASCRIPT

```javascript
// src/middleware/validate-message.js

const { ValidationError } = require('../errors/chat-errors');
const { MAX_MESSAGE_LENGTH } = require('../constants');

/**
 * Parse and validate raw WebSocket message string
 * @param {string} rawMessage - Raw message from WebSocket
 * @returns {WebSocketMessage} - Parsed and validated message object
 * @throws {ValidationError} - If message fails validation
 */
function parseAndValidateWebSocketMessage(rawMessage) {
  // TODO 1: Check if rawMessage is a string, throw ValidationError if not
  // TODO 2: Trim the message and check if empty, throw if empty
  // TODO 3: Try to parse JSON using JSON.parse, catch SyntaxError and throw ValidationError
  // TODO 4: Validate parsed object has required fields: type (string) and payload (object)
  // TODO 5: Validate type is one of allowed events (EVENT_CHAT_MESSAGE, EVENT_USER_JOINED, etc.)
  // TODO 6: Return the validated WebSocketMessage object
}

/**
 * Validate chat message payload structure
 * @param {object} payload - The payload from chat_message event
 * @returns {ChatMessage} - Validated chat message data
 * @throws {ValidationError} - If payload fails validation
 */
function validateChatMessagePayload(payload) {
  // TODO 1: Check payload is an object, throw if not
  // TODO 2: Validate required fields exist: sender (string), content (string), roomId (string)
  // TODO 3: Validate content length <= MAX_MESSAGE_LENGTH, throw if too long
  // TODO 4: Validate content is not just whitespace (trim().length > 0)
  // TODO 5: Validate sender is not empty string
  // TODO 6: Validate roomId matches pattern (alphanumeric, hyphens, underscores)
  // TODO 7: Return the validated payload (will be used to create ChatMessage)
}

/**
 * Enhanced message handler with comprehensive error handling
 * @param {string} connectionId - The connection ID
 * @param {string} rawMessage - Raw WebSocket message
 * @returns {void}
 */
```

```
function handleMessage(connectionId, rawMessage) {
  try {
    // TODO 1: Parse and validate the message using parseAndValidateWebSocketMessage
    // TODO 2: Check rate limit for this connection, throw RateLimitError if exceeded
    // TODO 3: Validate user session (call validateSession with token from connection)
    // TODO 4: Route to appropriate handler based on message.type
    // TODO 5: Catch any errors thrown during handler execution
  } catch (error) {
    // TODO 6: Log the error with appropriate level (warn for client errors, error for server errors)
    // TODO 7: If error is a ChatError, send formatted error response to client
    // TODO 8: If error is unexpected (TypeError, etc.), log with stack trace and send generic error
    // TODO 9: For certain errors (AuthenticationError), close the WebSocket connection
  }
}
```

Core Logic Skeleton: Connection Heartbeat Implementation

```javascript
// src/server/connection-manager.js

class ConnectionManager {
  constructor() {
    this.connections = new Map(); // connectionId -> ClientSession
    this.heartbeatInterval = 25000; // 25 seconds
    this.heartbeatTimeout = 30000; // 30 seconds timeout
    this.cleanupInterval = 60000; // Cleanup every minute
  }

  /**
   * Start heartbeat for a connection
   * @param {string} connectionId - The connection ID
   * @param {WebSocket} socket - The WebSocket instance
   */
  startHeartbeat(connectionId, socket) {
    // TODO 1: Create a ping interval that sends WebSocket ping every heartbeatInterval
    // TODO 2: Track last pong time in ClientSession (update on 'pong' event)
    // TODO 3: Set up timeout checker that verifies last pong was within heartbeatTimeout
    // TODO 4: If timeout exceeded, call this.removeConnection(connectionId, 'heartbeat_timeout')
  }

  /**
   * Periodic cleanup of stale connections
   */
  startCleanupJob() {
    setInterval(() => {
      const now = Date.now();
      for (const [connectionId, session] of this.connections) {
        // TODO 1: Check if session.lastActivity older than 60 seconds
        // TODO 2: Check if heartbeat timeout exceeded (last pong too old)
        // TODO 3: If stale, remove connection with reason 'inactive'
      }
    }, this.cleanupInterval);
  }
}
```

Language-Specific Hints for JavaScript/Node.js

1. **WebSocket Ping/Pong**: Use `ws` library's built-in support:

```javascript
const WebSocket = require('ws');

const wss = new WebSocket.Server({ clientTracking: true });



wss.on('connection', (ws) => {

  ws.isAlive = true;



  ws.on('pong', () => {

    ws.isAlive = true;

  });



  // Heartbeat interval

  const interval = setInterval(() => {

    if (ws.isAlive === false) {

      return ws.terminate();

    }

    ws.isAlive = false;

    ws.ping();

  }, 30000);



  ws.on('close', () => {

    clearInterval(interval);

  });

});
```

2. **Error Propagation**: Use async/await with try/catch:

```javascript
async function handleJoinRoom(connectionId, roomId) {
  try {
    const room = await roomManager.joinRoom(connectionId, roomId);
    sendToClient(connectionId, { type: 'room_joined', payload: room });
  } catch (error) {
    if (error instanceof ChatError) {
      // Expected error, send to client
      sendToClient(connectionId, createServerMessage('error', {
        code: error.code,
        message: error.message
      }));
    } else {
      // Unexpected, log and send generic error
      logger.error('Unexpected error in joinRoom', { error, connectionId });
      sendToClient(connectionId, createServerMessage('error', {
        code: 'INTERNAL_ERROR',
        message: 'Something went wrong'
      }));
    }
  }
}
```

3. **Rate Limiting with Redis** (advanced):

```javascript
const redis = require('redis');
const client = redis.createClient();

async function rateLimit(connectionId, limit = 100, windowMs = 60000) {
  const key = `rate_limit:${connectionId}`;
  const now = Date.now();
  const windowStart = now - windowMs;

  // Remove old timestamps
  await client.zremrangebyscore(key, 0, windowStart);

  // Count requests in current window
  const requestCount = await client.zcard(key);

  if (requestCount >= limit) {
    throw new RateLimitError(Math.ceil(windowMs / 1000));
  }

  // Add current request
  await client.zadd(key, now, now);
  await client.expire(key, Math.ceil(windowMs / 1000));

  return { remaining: limit - requestCount - 1 };
}
```

Debugging Tips for Error Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
Messages not appearing for some users	User not in room member list; connection dead but not cleaned	Check `Room.memberIds` contains connectionId; verify `lastActivity` timestamp	Ensure `joinRoom` updates member list; implement heartbeat cleanup
User count wrong (shows offline users)	Connection cleanup not running; zombie connections in member list	Log when connections are removed; check heartbeat is working	Implement periodic cleanup job; verify ping/pong events firing
Server crashes on malformed message	No try/catch in message handler; JSON.parse without validation	Add error handling wrapper; validate before parsing	Implement `parseAndValidateWebSocketMessage` with try/catch
Memory usage grows indefinitely	Connections not cleaned up; message history not paginated	Monitor connection count; check room cleanup job	Implement connection limits; paginate history queries
Users receive own messages	`broadcastToRoom` not excluding sender	Check `excludeConnectionId` parameter is passed correctly	Verify broadcast logic excludes sender connection
Typing indicator never disappears	Timeout not cleared; client not sending stop event	Log typing events; check timer cleanup on disconnect	Implement debouncing with clearTimeout; handle disconnect cleanup

Milestone Checkpoint for Error Handling

After implementing error handling, verify by:

1. **Test Network Failure Simulation**:

```bash
# Start server
npm start

# In browser, open chat and send messages
# Then simulate network drop (turn off Wi-Fi, or use browser DevTools to go offline)
# Wait 30+ seconds, then restore network
# Expected: Server should clean up user after ~60 seconds; user should auto-reconnect
```

2. **Test Malformed Messages**:

```bash
# Use WebSocket CLI tool or browser console
const ws = new WebSocket('ws://localhost:3000');
ws.onopen = () => {
  ws.send('not json'); // Should get error response
  ws.send('{"type":"chat_message","payload":{}}'); // Missing fields
  ws.send('A'.repeat(2000)); // Too long
};
# Expected: Server responds with error messages, doesn't crash
```

3. **Verify Rate Limiting**:

```bash
# Script to send 200 messages quickly
# Expected: After ~100 messages, get rate limit error
```

4. **Check Memory Usage**:

```bash
# Monitor Node.js process
node -e "setInterval(() => console.log(process.memoryUsage()), 1000)" &
# Connect many clients, send messages, disconnect
# Expected: Memory returns to baseline after connections close
```

Signs of correct implementation:

- Server remains running despite invalid inputs
- Error messages are descriptive and help debug
- Connections are cleaned up automatically
- Memory usage stabilizes over time

Testing Strategy

> **Milestone(s):** All milestones (1-4). Testing is an integral part of building a reliable real-time chat system. This section provides a practical, learning-focused approach to verifying your implementation works correctly, with emphasis on integration testing and manual verification checkpoints aligned with each milestone.

Testing a real-time WebSocket application presents unique challenges compared to traditional HTTP APIs. Connections are stateful, events occur asynchronously, and timing issues can create subtle bugs. This testing strategy balances rigor with practicality for learners, focusing on **integration testing** (verifying components work together) and **manual verification** (observing system behavior) while providing a foundation for more advanced testing patterns.

Testing Approach and Property Verification

Mental Model: The Airport Control Tower and Flight Simulator Think of testing this chat system like operating an airport control tower. **Unit tests** are like checking individual aircraft systems in isolation (fuel gauge, landing gear). **Integration tests** are like simulating takeoff and landing procedures with multiple systems interacting. **Manual verification** is like the control tower operator watching actual flights move on radar, confirming the whole system works as expected. For learning, we prioritize the control tower view (integration) while ensuring critical individual systems (units) function correctly.

Testing Layers and Focus Areas

Our testing pyramid for this project has three layers with increasing realism but decreasing automation:

Layer	What It Tests	How We'll Implement	Primary Goal
Unit Tests	Individual functions and classes in isolation	Jest/Node.js test files mocking dependencies	Verify core logic (message parsing, validation, business rules)
Integration Tests	Components working together (e.g., ConnectionManager + RoomManager)	Test server with in-memory WebSocket clients	Verify event flow and state synchronization
Manual Verification	End-to-end system behavior with real browsers	Multiple browser tabs connecting to your running server	Build intuition and catch UI/experience issues

For this learning project, we emphasize **integration testing** because WebSocket systems derive most bugs from component interactions rather than isolated logic. However, we'll write unit tests for critical validation and transformation functions.

Key Properties to Verify

These are the fundamental guarantees your system must maintain. Each property can be tested through specific scenarios:

Property	Why It Matters	How to Test
Message Ordering	Users expect messages to appear in the order they were sent, not delivery order	Send messages M1, M2, M3 from same client; verify receivers see same sequence
At-Most-Once Delivery	Messages shouldn't duplicate, causing confusing repeated messages	Send same message rapidly multiple times; verify only one appears
User Removal on Disconnect	Prevent "ghost users" who appear online but disconnected	Connect user, verify in member list, disconnect, verify removed within 5 seconds
Room Isolation	Messages in Room A must not leak to Room B	Create two rooms, send message in Room A, verify users in Room B don't receive it
History Integrity	New users should see accurate past conversation	Send messages, disconnect all users, new user joins, should see same messages
Presence Accuracy	Typing indicators and online status must reflect current reality	User starts typing, indicator appears; stops typing for 3 seconds, indicator disappears
Authentication Enforcement	Unauthenticated users cannot send/receive messages	Try to send message without valid session; connection should be rejected or ignored
Connection Resilience	Network drops shouldn't break application state	Disconnect network for 10 seconds, reconnect, verify automatic rejoin and state recovery

Integration Testing Strategy

For integration tests, we'll simulate the complete system with these components:

1. **Test Server Instance**: A real WebSocket server running on a test port
2. **Mock Clients**: Programmatic WebSocket clients that can send/receive messages
3. **Test Orchestrator**: Code that coordinates scenarios and asserts outcomes

The test pattern follows this sequence:

```
1. Start test server with clean state
2. Connect mock Client A and Client B via WebSocket
3. Authenticate both clients (simulate login)
4. Have both clients join the same room
5. Client A sends a chat message
6. Verify Client B receives the message with correct fields
7. Clean up connections and server
```

> **Design Insight**: Integration tests for WebSocket systems are inherently stateful and asynchronous. Use explicit timeouts and await patterns rather than fixed sleeps. Each test should clean up completely to prevent test pollution.

ADR: Choosing Integration Tests Over Unit Tests for Core Components

> **Decision: Prioritize Integration Testing for Component Interactions**
>
> - **Context**: WebSocket chat systems involve complex interactions between connection management, room membership, and message routing. Bugs often emerge at component boundaries rather than within isolated functions.
> - **Options Considered**:
>   1. **Primarily unit tests** with extensive mocking of WebSocket and database dependencies
>   2. **Balanced approach** with unit tests for pure functions and integration tests for components
>   3. **Primarily integration tests** with real WebSocket connections and in-memory stores
> - **Decision**: Option 3 (primarily integration tests) for core components, with unit tests only for pure validation/utility functions
> - **Rationale**:
>   - Integration tests verify the actual event flow and message passing that unit tests with mocks might miss
>   - Real WebSocket behavior (connection drops, buffering) is difficult to mock accurately
>   - For learning, seeing the whole system work builds better intuition than passing unit tests with mocked dependencies
>   - The relatively small codebase makes integration test execution fast enough
> - **Consequences**:
>   - Tests are more realistic but potentially more brittle to timing issues
>   - Test setup/teardown is more complex (requires starting actual server)
>   - Encourages designing components with clear interfaces for testability

Option	Pros	Cons	Why Not Chosen
Primarily unit tests	Fast execution, isolated failures	Misses integration issues, complex mocks	Doesn't test real WebSocket behavior
Balanced approach	Tests both logic and integration	More test code to maintain, context switching	Good but heavier for learning focus
Primarily integration tests	**Tests real interactions, builds system intuition**	**Slower, timing-sensitive**	**Chosen: Best for learning real-time system behavior**

Milestone Implementation Checkpoints

Each milestone has clear acceptance criteria. These checkpoints help you verify your implementation meets them before moving forward. Think of these as **flight checks** before taking off to the next milestone.

Milestone 1: WebSocket Server Setup Checkpoint

Expected Behavior: Your server accepts WebSocket connections, tracks active clients, handles disconnections, and doesn't crash on malformed messages.

Verification Steps:

1. **Start the server**:

```bash
node server.js
```

Expected output: "WebSocket server listening on port 8080" (or your chosen port)

2. **Test basic connection** using browser DevTools:
   - Open Chrome/Firefox, press F12, go to Console tab
   - Create WebSocket connection:

```javascript
const ws = new WebSocket('ws://localhost:8080');

ws.onopen = () => console.log('Connected!');

ws.onmessage = (event) => console.log('Received:', event.data);
```

- Expected: "Connected!" appears in console
- Check server logs: Should show "New connection from [IP]"

3. **Verify connection tracking**:
   - Open 3 browser tabs, run connection code in each
   - Send a test message from one client:

```javascript
ws.send(JSON.stringify({type: 'ping', payload: 'test'}));
```

- Check server logs: Should show message received and handled
- Close one tab, check server logs within 2 seconds: Should show "Connection closed" or similar

4. **Test error resilience**:
   - With a client connected, send invalid JSON:

```javascript
ws.send('{ malformed json }');
```

- Expected: Server doesn't crash, may send error response or close connection gracefully
- Send extremely long message (>1MB):

```javascript
ws.send('x'.repeat(1000000));
```

- Expected: Server rejects or handles without memory issues

Success Indicators:

- ✅ Server starts without errors
- ✅ Multiple clients can connect simultaneously
- ✅ Server logs show connect/disconnect events
- ✅ Invalid messages don't crash server
- ✅ All connections cleaned up when tabs close

Common Pitfalls at This Stage:

- ⚠️ **Port already in use**: Change port or kill previous process
- ⚠️ **No server logs for connections**: Check WebSocket upgrade handling in HTTP server
- ⚠️ **Connections never show as closed**: Ensure `socket.on('close')` handler removes from connection tracking

Milestone 2: Message Broadcasting Checkpoint

Expected Behavior: Messages broadcast to all room members, include proper metadata (sender, timestamp), and typing indicators work.

Verification Steps:

1. **Set up test scenario** with three browser tabs (User A, User B, Observer):
   - Each tab connects and joins the same default room (implement simple join for now)
   - Use different usernames in each tab's connection
2. **Test message broadcasting**:
   - User A sends: `{"type": "chat_message", "payload": {"content": "Hello everyone"}}`
   - Expected: User B and Observer receive message with:
     - Correct content
     - Sender field showing User A's username
     - Recent timestamp (within last second)
     - User A should NOT receive their own message back
3. **Test typing indicators**:
   - User A sends: `{"type": "typing_start", "payload": {"roomId": "default"}}`
   - Expected: User B and Observer receive typing indicator with User A's username
   - Wait 3 seconds: User B and Observer should receive `typing_stop` automatically
   - Alternative: User A sends `typing_stop` manually after 1 second
4. **Test join/leave notifications**:
   - Open fourth tab (User D), join room
   - Expected: Users A, B, Observer receive `user_joined` notification
   - User D leaves (close tab or send leave message)
   - Expected: Remaining users receive `user_left` within 5 seconds

Success Indicators:

- ✅ Messages appear in all other clients but not sender
- ✅ Each message has sender, content, and ISO timestamp
- ✅ Typing indicators appear and clear automatically
- ✅ Join/leave events trigger notifications
- ✅ Server logs show broadcast counts (e.g., "Broadcasting to 2 clients")

Advanced Verification Script: Create a simple Node.js test script that automates this verification:

```javascript
// test-broadcast.js - Basic verification script

const WebSocket = require('ws');

// Implementation details would go here
```

Milestone 3: Chat Rooms Checkpoint

Expected Behavior: Users can create/join rooms, messages are room-scoped, member lists update correctly, and room listing works.

Verification Steps:

1. **Test room creation and joining**:
   - User A sends: `{"type": "join_room", "payload": {"roomName": "gaming"}}`
   - Expected: Server creates "gaming" room, User A receives `room_joined` confirmation
   - User B sends join request for same room
   - Expected: User B joins, User A receives `user_joined` notification for User B
2. **Verify room isolation**:
   - User A in "gaming" room sends message
   - User C in "general" room (different room) should NOT receive it
   - User B in "gaming" room SHOULD receive it
3. **Test member list accuracy**:
   - User D joins "gaming" room
   - User A requests member list: `{"type": "get_members", "payload": {"roomId": "gaming"}}`
   - Expected: Response includes User A, B, D (all current members)
   - User B leaves room
   - User A requests member list again within 5 seconds: Should NOT include User B
4. **Test room listing**:
   - Create rooms "gaming", "music", "random"
   - Request room list: `{"type": "get_rooms", "payload": {}}`
   - Expected: All three rooms listed with correct member counts
   - Empty room after all users leave should either be removed or show 0 members
5. **Test message history**:
   - Send 5 messages in "gaming" room
   - New user joins "gaming" room
   - Expected: New user receives last 3-5 messages (configurable) immediately after joining

Success Indicators:

- ✅ Room-specific messaging works (no cross-room leakage)
- ✅ Member lists accurately reflect current occupants
- ✅ Room listing shows all active rooms with counts
- ✅ New room members receive recent message history
- ✅ Empty rooms don't accumulate or are cleaned up

Milestone 4: User Authentication & Persistence Checkpoint

Expected Behavior: Users authenticate before chatting, messages persist to database, history loads with pagination, and presence works across reconnections.

Verification Steps:

1. **Test authentication requirement**:

- Try to connect WebSocket without valid session token
- Expected: Connection rejected during HTTP upgrade or immediately closed with error
- Properly authenticate via HTTP login first, then connect with session token

2. **Verify message persistence**:
   - Send 10 messages in a room
   - Restart the server completely (simulating crash)
   - Reconnect clients, rejoin room
   - Expected: Previous 10 messages still available via history
   - New messages should continue with new IDs/timestamps
3. **Test history pagination**:
   - Send 25 messages to a room
   - New user joins, requests history with limit=10
   - Expected: Receives most recent 10 messages, not first 10
   - Request older messages with `beforeId` parameter
   - Expected: Next 10 older messages
4. **Verify presence across sessions**:
   - User A logs in on Browser Tab 1 (shows as online)
   - Same user logs in on Browser Tab 2
   - Expected: Other users see User A as online (single user, multiple connections)
   - Close Tab 1, User A should remain online (Tab 2 still active)
   - Close Tab 2, within 10 seconds User A shows as offline
5. **Test typing indicator persistence**:
   - User A starts typing in room
   - User B joins room mid-typing
   - Expected: User B immediately sees User A's typing indicator
   - User A stops typing, both User B and new user see indicator clear

Success Indicators:

- ✅ Unauthenticated connections rejected
- ✅ Messages survive server restart
- ✅ History pagination works (most recent first)
- ✅ User presence accurate across multiple devices/sessions
- ✅ Typing indicators synchronized correctly

> **Testing Pro Tip**: Create a **test harness** with these exact verification steps as commented code. As you implement each milestone, uncomment and run the corresponding test block. This builds your test suite incrementally while ensuring backward compatibility.

Implementation Guidance

This guidance provides practical code structures and test utilities to implement the testing strategy.

A. Technology Recommendations Table

Component	Simple Option (Learning)	Advanced Option (Production)
Test Runner	Node.js built-in `assert` + manual scripts	Jest testing framework
WebSocket Client for Tests	`ws` npm package (same as server)	Puppeteer for browser automation
Mock/Stub Library	Manual mock objects	Jest auto-mocking or Sinon.js
Test Coverage	Manual verification checkpoints	Istanbul/NYC for coverage reports
Continuous Testing	Manual re-running of tests	GitHub Actions with automated test suite

B. Recommended File/Module Structure

```
real-time-chat/
├── src/
│   ├── server.js                    # Main server entry point
│   ├── connection-manager.js        # Milestone 1 component
│   ├── room-manager.js              # Milestone 2-3 component
│   ├── auth-service.js              # Milestone 4 component
│   └── message-store.js             # Milestone 4 persistence
├── tests/
│   ├── integration/                  # Integration tests
│   │   ├── connection-manager.test.js
│   │   ├── room-manager.test.js
│   │   └── auth-integration.test.js
│   ├── unit/                        # Unit tests for pure functions
│   │   ├── validation.test.js
│   │   ├── message-format.test.js
│   │   └── crypto-utils.test.js
│   └── utils/
│       ├── test-client.js           # Reusable WebSocket test client
│       └── test-server.js           # Test server setup/teardown
├── scripts/
│   ├── test-milestone1.js           # Manual verification scripts
│   ├── test-milestone2.js
│   └── benchmark-connections.js     # Optional load testing
└── package.json
```

C. Test Infrastructure Starter Code

Here's a complete, reusable test client utility for integration tests:

JAVASCRIPT

```javascript
// tests/utils/test-client.js

const WebSocket = require('ws');


/**

 * TestWebSocketClient - Programmatic client for integration testing

 * Wraps WebSocket with promise-based methods for easier testing

 */

class TestWebSocketClient {

  constructor(url) {

    this.url = url;

    this.socket = null;

    this.messageQueue = [];

    this.messageHandlers = new Map();

    this.connectPromise = null;

  }


  /**

   * Connect to the server with optional authentication token

   */

  connect(token = null) {

    const url = token ? `${this.url}?token=${token}` : this.url;

    this.socket = new WebSocket(url);



    this.connectPromise = new Promise((resolve, reject) => {

      this.socket.on('open', () => {

        console.log(`Test client connected to ${url}`);

        resolve();

      });



      this.socket.on('error', (error) => {

        reject(new Error(`Connection failed: ${error.message}`));

      });



      this.socket.on('message', (rawData) => {

        try {

          const message = JSON.parse(rawData.toString());

          this.messageQueue.push(message);



          // Call any registered handlers for this message type

          const handlers = this.messageHandlers.get(message.type) || [];

          handlers.forEach(handler => handler(message));
```

```
      } catch (error) {
        console.error('Failed to parse message:', rawData.toString());
      }
    });
  });

  return this.connectPromise;
}

/**
 * Send a JSON message to the server
 */
send(type, payload) {
  if (!this.socket || this.socket.readyState !== WebSocket.OPEN) {
    throw new Error('WebSocket not connected');
  }

  const message = { type, payload, timestamp: new Date().toISOString() };
  this.socket.send(JSON.stringify(message));
  return message;
}

/**
 * Wait for a specific message type with timeout
 */
waitForMessage(messageType, timeoutMs = 5000) {
  return new Promise((resolve, reject) => {
    // Check if message already in queue
    const existing = this.messageQueue.find(m => m.type === messageType);
    if (existing) {
      resolve(existing);
      return;
    }

    // Set up handler for future message
    const handler = (message) => {
      if (message.type === messageType) {
        clearTimeout(timeoutId);
        resolve(message);
      }
    };
```

```
      // Add to handlers
      if (!this.messageHandlers.has(messageType)) {
        this.messageHandlers.set(messageType, []);
      }
      this.messageHandlers.get(messageType).push(handler);

      // Set timeout
      const timeoutId = setTimeout(() => {
        const handlers = this.messageHandlers.get(messageType) || [];
        const index = handlers.indexOf(handler);
        if (index > -1) handlers.splice(index, 1);
        reject(new Error(`Timeout waiting for message type: ${messageType}`));
      }, timeoutMs);
    });
  }

  /**
   * Get all received messages of a specific type
   */
  getMessagesOfType(messageType) {
    return this.messageQueue.filter(m => m.type === messageType);
  }

  /**
   * Clear the message queue
   */
  clearQueue() {
    this.messageQueue = [];
  }

  /**
   * Close the connection
   */
  close() {
    if (this.socket) {
      this.socket.close();
      this.socket = null;
    }
    this.messageHandlers.clear();
  }
}
```

```
module.exports = TestWebSocketClient;
```

D. Core Test Skeleton Code

Here are test skeletons for key integration scenarios:

JAVASCRIPT

```javascript
// tests/integration/room-manager.test.js

const TestWebSocketClient = require('../utils/test-client');

const { startTestServer, stopTestServer } = require('../utils/test-server');


describe('Room Manager Integration Tests', () => {

  let server;

  let clientA, clientB, clientC;

  const TEST_PORT = 8081;


  beforeAll(async () => {

    // Start test server on separate port

    server = await startTestServer(TEST_PORT);

  });


  afterAll(async () => {

    await stopTestServer(server);

  });


  beforeEach(async () => {

    // Create fresh clients for each test

    clientA = new TestWebSocketClient(`ws://localhost:${TEST_PORT}`);

    clientB = new TestWebSocketClient(`ws://localhost:${TEST_PORT}`);

    clientC = new TestWebSocketClient(`ws://localhost:${TEST_PORT}`);



    // TODO 1: Connect all clients with valid authentication tokens

    // TODO 2: Set up default usernames for each client

  });


  afterEach(() => {

    // Clean up clients

    if (clientA) clientA.close();

    if (clientB) clientB.close();

    if (clientC) clientC.close();

  });


  test('Messages broadcast to all room members except sender', async () => {

    // TODO 3: Have clientA and clientB join the same room

    // TODO 4: Have clientC join a different room

    // TODO 5: ClientA sends a chat message to the room

    // TODO 6: Verify clientB receives the message with correct content and sender

    // TODO 7: Verify clientC does NOT receive the message (different room)

    // TODO 8: Verify clientA does NOT receive its own message back

  });
```

```
  test('User join/leave notifications work correctly', async () => {
    // TODO 9: ClientA joins "test-room"
    // TODO 10: ClientB joins "test-room"
    // TODO 11: Verify clientA receives user_joined notification for clientB
    // TODO 12: ClientB leaves the room
    // TODO 13: Verify clientA receives user_left notification for clientB
  });

  test('Message history loads when joining room', async () => {
    // TODO 14: ClientA joins "history-room" and sends 5 messages
    // TODO 15: ClientB joins "history-room"
    // TODO 16: Verify clientB receives the 5 historical messages (or configured limit)
    // TODO 17: Verify messages are in correct chronological order
  });
});
```

E. Language-Specific Hints (JavaScript/Node.js)

- **Async Testing**: Use `async/await` with proper timeouts. Jest supports async tests natively.
- **WebSocket Testing**: The `ws` package works in both server and test environments. Remember that WebSocket events are asynchronous.
- **Mocking**: For unit tests, create simple factory functions that return mock objects:

```javascript
const createMockSocket = () => ({
  send: jest.fn(),
  close: jest.fn(),
  readyState: 1, // OPEN
  on: jest.fn()
});
```

- **Cleaning Up**: Always close WebSocket connections after tests to avoid port exhaustion.

F. Milestone Checkpoint Verification Scripts

Create these scripts to manually verify each milestone:

JAVASCRIPT

```javascript
// scripts/test-milestone1.js - Basic connection verification

const WebSocket = require('ws');

const readline = require('readline');


const rl = readline.createInterface({

  input: process.stdin,

  output: process.stdout

});


console.log('=== Milestone 1: WebSocket Server Verification ===\n');


const ws = new WebSocket('ws://localhost:8080');


ws.on('open', function open() {

  console.log('✅ Connected to server');

  console.log('1. Connection established');



  // Test message sending

  ws.send(JSON.stringify({ type: 'ping', payload: 'test' }));

  console.log('2. Sent test message');



  // Wait for any response

  setTimeout(() => {

    console.log('3. Connection active for 3 seconds');



    rl.question('\nPress Enter to close connection and verify server cleanup...', () => {

      ws.close();

      console.log('Connection closed. Check server logs for cleanup message.');

      console.log('\n✅ Milestone 1 Verification Complete');

      console.log('Checklist:');

      console.log('  - Server accepted connection');

      console.log('  - Messages can be sent');

      console.log('  - Server handles disconnection');

      rl.close();

    });

  }, 3000);

});


ws.on('message', function incoming(data) {

  console.log(`📨 Received: ${data}`);

});


ws.on('error', function error(err) {
```

```
  console.error('❌ Connection error:', err.message);

  rl.close();

});


ws.on('close', function close() {

  console.log('🔌 Connection closed by server or client');

});
```

G. Debugging Tips for Testing

Symptom	Likely Cause	How to Diagnose	Fix
Messages not appearing	Room membership incorrect or broadcast logic flawed	1. Check server logs for broadcast count 2. Verify receiver is in same room as sender 3. Check message payload structure	Ensure `RoomManager.broadcastToRoom` excludes sender and sends to correct member list
User count wrong after disconnect	Connection not removed from room membership on disconnect	1. Check if `ConnectionManager.removeConnection` calls `RoomManager.leaveAllRooms` 2. Verify disconnection detection timing	Add explicit cleanup in connection close handler
Server crashes on invalid JSON	Missing try-catch in message parsing	1. Look for unhandled exception in logs 2. Check `parseAndValidateWebSocketMessage` error handling	Wrap JSON.parse in try-catch, send error response instead of crashing
Typing indicators never clear	Missing timeout or stop event not sent	1. Check if `setUserTyping` sets/resets timers 2. Verify client sends `typing_stop` on blur	Implement server-side timeout (3 seconds) as backup to client events
History loads wrong messages	Database query order incorrect or pagination flawed	1. Check `getMessagesForRoom` ORDER BY clause 2. Verify limit and offset calculations	Sort by timestamp DESC for most recent first, use cursor-based pagination
Authentication bypass possible	WebSocket upgrade doesn't validate session	1. Try connecting without token 2. Check `authenticateConnection` logic	Validate session during HTTP upgrade before accepting WebSocket connection

Debugging Guide

> **Milestone(s):** All milestones (1-4)

Even the most carefully designed systems encounter bugs during implementation. This section provides a practical handbook for diagnosing and fixing the most common issues learners face when building this real-time chat system. Think of debugging as detective work: you observe symptoms, form hypotheses about causes, gather evidence through testing, and apply fixes based on your understanding of the system architecture.

When building WebSocket applications, bugs often manifest in subtle ways because of the asynchronous, stateful nature of persistent connections. A bug in connection cleanup might not appear until hours of operation, when memory usage balloons. A race condition in room membership might only surface when two users join simultaneously. This guide helps you recognize these patterns early and provides systematic approaches to isolate and fix them.

Common Bugs: Symptom → Cause → Fix

The following table catalogs the most frequent issues learners encounter, organized by symptom area. Each entry includes the observable symptom, the likely underlying cause, and step-by-step diagnosis steps leading to a fix.

Symptom	Likely Cause	Diagnosis Steps	Fix
Messages not appearing for other users (but sender sees their own)	The `broadcastToRoom` method is including the sender in the broadcast, or the room membership tracking has incorrect data.	1. Check the `RoomManager.broadcastToRoom` implementation - is it using `excludeConnectionId` parameter? 2. Log the list of connection IDs in the room before broadcasting. 3. Verify the sender's connection ID matches their actual connection ID in the room's member set.	Ensure `broadcastToRoom` iterates through `room.memberIds` and skips the `excludeConnectionId`. Verify room membership is updated correctly on join/leave.
User count wrong in room listing (shows offline users or misses online ones)	Connection cleanup not removing users from rooms on disconnect, or race condition between join/leave events.	1. Add logging to `ConnectionManager.removeConnection` to see when it's called. 2. Verify `RoomManager.leaveAllRooms(connectionId)` is called in disconnect handler. 3. Check for duplicate connection IDs or users joining multiple times.	Implement `ConnectionManager.removeConnection` that calls `RoomManager.leaveAllRooms`. Add connection ID uniqueness validation.
Server crashes when client disconnects	Trying to send to or access a socket that has already closed, missing error handlers on WebSocket operations.	1. Look for `socket.send()` calls without checking `socket.readyState === WebSocket.OPEN`. 2. Check if event listeners on closed sockets still fire. 3. Look for null/undefined socket references in connection tracking map.	Wrap all `socket.send()` calls with `if (socket.readyState === WebSocket.OPEN)`. Implement `ConnectionManager.sendToClient` with this check.
Typing indicators never disappear	No timeout to clear typing status, or timeout not triggering due to event loop issues.	1. Check `RoomManager.setUserTyping` implementation - is there a `setTimeout` to clear? 2. Verify the timeout function has access to current connection ID and room ID. 3. Check if multiple timeouts are created without clearing previous ones.	Implement debouncing: store timeout ID per user, clear previous timeout before setting new one. Use `clearTimeout()` in `setUserTyping`.
Old messages appear multiple times when rejoining room	Message history loading logic doesn't account for messages already loaded, or pagination cursor logic flawed.	1. Check `getRecentMessagesForRoom` - is it always returning same N messages? 2. Verify client isn't storing and re-displaying old messages. 3. Check if message IDs are duplicated in database.	Implement message pagination with `lastMessageId` parameter. Client should track highest message ID received.
Can't join room with special characters in name	Room name not sanitized, causing issues with room ID generation or storage key creation.	1. Check `createRoom` function - how does it generate room ID from name? 2. Look for URL-unsafe characters in WebSocket event payloads. 3. Test with room names containing spaces, slashes, or Unicode.	Sanitize room names: convert to lowercase, replace spaces with hyphens, remove special characters. Use a separate generated ID for internal storage.
Memory usage grows continuously	Connection objects not cleaned up on disconnect, or message history accumulating without bounds.	1. Monitor `ConnectionManager.connections` map size over time. 2. Check if `Room.memberIds` sets grow without removal. 3. Verify message store cleanup (`deleteMessagesOlderThan`) is called.	Implement periodic cleanup of orphaned rooms. Add connection timeout for inactive sockets. Ensure all remove/disconnect paths clean up state.
Authentication fails even with correct credentials	Password hash comparison failing (timing attack protection), or session token not being properly attached to WebSocket upgrade.	1. Verify password hashing uses same salt rounds and algorithm. 2. Check if authentication token is included in WebSocket upgrade URL. 3. Test with a known password hash to isolate hashing issue.	Use constant-time comparison for password hashes. Ensure token is passed in `Sec-WebSocket-Protocol` header or query parameter.
Messages arrive out of order	No timestamp or sequence number in message protocol, asynchronous broadcasting causing race conditions.	1. Check if `ChatMessage.timestamp` uses client time instead of server time. 2. Look for parallel `socket.send()` operations without queuing. 3. Verify messages are stored with monotonic increasing IDs.	Assign server-side timestamps in `createChatMessage`. Use database auto-increment ID for ordering. Implement client-side message queue.
Rapid clicking causes	No client-side debouncing on send button, and no idempotency token in message	1. Observe network traffic in browser dev tools - are duplicate WebSocket frames sent?	Add client-side debouncing (disable button for 500ms). Include unique message ID in payload,

Symptom	Likely Cause	Diagnosis Steps	Fix
duplicate messages	protocol.	2. Check if UI disables send button during transmission. 3. Verify server doesn't process same message ID twice.	reject duplicates on server.
Server becomes unresponsive after many connections	No connection limits, no heartbeat to detect dead connections, synchronous operations blocking event loop.	1. Monitor CPU and memory usage during load test. 2. Check for synchronous file I/O or JSON parsing of large messages. 3. Verify heartbeat (ping/pong) is implemented and timing out dead connections.	Implement WebSocket ping/pong. Add connection limit per IP. Use asynchronous JSON parsing with try-catch.
Users see other users' messages from wrong rooms	Message routing bug - messages sent to all rooms or wrong room ID in broadcast.	1. Check `ChatMessage.roomId` matches intended room. 2. Verify `RoomManager.broadcastToRoom` is called with correct roomId. 3. Look for global broadcast instead of room-specific broadcast.	Double-check room assignment in message handler. Add validation that sender is in the room they're messaging.
"Welcome" messages appear for users already in room	Join notification being sent to the joining user themselves, not just other members.	1. Check `EVENT_USER_JOINED` broadcast logic - is sender excluded? 2. Verify separate welcome message isn't being sent to new user. 3. Test with two clients - who receives the join notification?	Ensure `broadcastToRoom` for join notifications uses `excludeConnectionId` parameter. Send separate `room_joined` event to new user.
Browser tab duplication causes ghost users	Same session token used in multiple tabs, causing duplicate presence entries.	1. Open two tabs with same login - do both appear as separate users? 2. Check if session validation allows multiple connections. 3. Look for user ID to connection ID mapping issues.	Implement one-session-per-connection model. On new connection, disconnect previous connection with same user ID.
Server crashes on malformed JSON message	No try-catch in `parseAndValidateWebSocketMessage`, raw message parsing throws exception.	1. Send invalid JSON via WebSocket client and observe crash. 2. Check if error propagates to connection error handler. 3. Look for missing validation in message type and payload.	Wrap JSON.parse in try-catch, send error message to client, keep connection alive. Validate all required fields exist and are correct type.
Clock shows different times for same message	Using client-provided timestamps instead of server-assigned timestamps.	1. Compare `ChatMessage.timestamp` in database vs. what client displays. 2. Check if client is overriding server timestamp with local time. 3. Verify all clients use same timezone display.	Always use server-generated ISO timestamps in `createChatMessage`. Client should display relative time ("2 minutes ago") rather than absolute.
Typing indicator shows for disconnected users	No cleanup of typing status when user disconnects or leaves room.	1. Disconnect while typing indicator is active - does it persist? 2. Check `RoomManager.leaveRoom` - does it clear typing status? 3. Verify timeout clears typing status even if disconnect happens first.	Call `setUserTyping(connectionId, roomId, false)` in `leaveRoom` and `leaveAllRooms`. Store typing status per connection, not per user.

Debugging Techniques and Tools

Effective debugging requires both systematic thinking and the right tools. Below are techniques specifically valuable for WebSocket-based real-time systems.

Strategic Logging: The Time-Traveling Observer

> Add logs as if you're creating a flight recorder for your application. Each log entry should include: timestamp, connection ID, user ID (if known), room ID (if relevant), and the action being performed.

Create a logging helper that wraps key operations:

1. **Connection lifecycle logging**: Log when connections open, authenticate, join rooms, send messages, and close. Include the reason for closure (normal, error, timeout).
2. **State dumps**: Periodically log the size of connection maps, room member counts, and memory usage. This helps identify leaks.
3. **Message flow tracing**: Assign each message a unique correlation ID and log it at each processing stage (receive, validate, route, store, broadcast).

Example logging strategy table:

When to Log	What to Include	Why It Helps
On WebSocket upgrade	`connectionId`, client IP, userAgent	Track connection origins and identify problematic clients
Before/after `broadcastToRoom`	`roomId`, `messageId`, recipient count	Verify messages reach intended recipients
On room join/leave	`connectionId`, `userId`, `roomId`, member count before/after	Debug room membership issues
On heartbeat timeout	`connectionId`, last activity timestamp	Identify zombie connections
On error in message handler	`connectionId`, raw message, error stack	Catch malformed messages before they crash server

Browser WebSocket Inspector: The Conversation Eavesdropper

Modern browser developer tools include WebSocket inspectors that let you observe the raw messages flowing between client and server.

Using Chrome DevTools:

1. Open DevTools (F12) → Network tab
2. Refresh page to establish WebSocket connection
3. Click the WebSocket request (ws:// or wss://)
4. View "Messages" tab to see frames in real-time
5. Filter by message type or search content

What to look for:

- Are messages being sent when you expect them?
- Are the message formats correct (valid JSON)?
- Is the server responding with expected acknowledgments?
- Are there error messages from the server?
- Is the connection closing unexpectedly?

Pro tip: You can manually send WebSocket messages from the Console in some browsers using `ws.send(JSON.stringify(...))` to test server responses.

Network Failure Simulation: The Chaos Engineer

Many bugs only appear under network instability. Simulate these conditions systematically:

Technique	How to Simulate	What Bugs It Reveals
Network delay	Use browser DevTools "Network Throttling" or a proxy like Charles	Timeout handling, message ordering issues
Packet loss	Use `tc` command on Linux: `tc qdisc add dev eth0 root netem loss 10%`	Message loss, retry logic, heartbeat failures
Connection interruption	Temporarily disable WiFi or use browser's "Offline" mode	Reconnection logic, state recovery
Server restart	Kill and restart server while clients connected	Session recovery, message persistence

Systematic testing approach:

1. Establish normal chat session with 2-3 clients
2. Apply network degradation
3. Observe behavior (do messages queue? do connections recover?)
4. Restore network
5. Verify system returns to normal state

The Rubber Duck Debugging Variant: Explain Your Architecture

When stuck, explain your implementation out loud or in writing, focusing on data flow:

1. "When Alice sends a message to room 'general':"
2. "Her browser calls `ws.send()` with this JSON structure..."
3. "The server receives it in `ConnectionManager.handleMessage()`..."
4. "It validates using `parseAndValidateWebSocketMessage()`..."
5. "Then calls `RoomManager.broadcastToRoom()` with these parameters..."
6. "Which retrieves the member set from the Room object..."
7. "And for each member except Alice, calls `ConnectionManager.sendToClient()`..."
8. "Which checks `socket.readyState` before sending..."

Often, the act of verbalizing the expected flow reveals where your implementation diverges.

State Visualization: The Whiteboard Method

Draw the current state of your system on a whiteboard or paper:

1. **Connection map**: Draw boxes for each connection ID, with arrows to user IDs and room IDs
2. **Room membership**: Create a table with rooms as columns and member connections as rows
3. **Message flow**: Trace a specific message through the system with timestamps

This is especially helpful for:

- Understanding why a particular user didn't receive a message
- Identifying memory leak sources (connections that should have been cleaned up)
- Visualizing race conditions in join/leave sequences

Differential Debugging: The Comparison Technique

When something works in one scenario but not another, systematically compare:

1. **Working vs. failing case**: What's different? Different room? Different user? Different message content?
2. **Before vs. after a change**: What code changed between when it worked and now?
3. **Local vs. production**: Environment differences (timeouts, memory limits, network config)

Create a checklist of comparison points:

- ☐ Connection parameters (authentication token present?)
- ☐ Room state (does room exist? correct member set?)
- ☐ Message format (all required fields present?)
- ☐ Server logs (any errors in one case but not the other?)
- ☐ Client state (any JavaScript errors in console?)

Implementation Guidance

This implementation guidance provides concrete code and techniques to implement the debugging approaches described above.

A. Technology Recommendations Table

Debugging Need	Simple Option	Advanced Option
Logging	`console.log` with timestamps	Winston or Pino with structured JSON logging
Network Inspection	Browser DevTools	Wireshark or `ws` library packet logger
Memory Profiling	Node.js `--inspect` flag + Chrome DevTools	Clinic.js or New Relic
Load Testing	Manual testing with multiple browser tabs	Artillery or k6 for WebSocket load testing
State Visualization	Manual drawing on whiteboard	Custom admin dashboard showing real-time connections

B. Debug Helper Module Structure

Create a dedicated debug module to keep instrumentation code organized:

```
project-root/
  src/
    server/
      connection-manager.ts    # Main WebSocket server logic
      room-manager.ts          # Room and presence management
      debug/                   # Debugging utilities
        logger.ts              # Structured logging helper
        state-inspector.ts     # Functions to dump current state
        test-client.ts         # Programmatic test WebSocket client
      index.ts                # Server entry point
```

C. Complete Debug Logger Implementation

Here's a complete, ready-to-use structured logger:

JAVASCRIPT

```javascript
// src/server/debug/logger.js

/**
 * Structured logger for debugging WebSocket chat application
 * Usage: logger.log('connect', { connectionId, userId, ip })
 */

class DebugLogger {
  constructor(enableConsole = true, enableFile = false) {
    this.enableConsole = enableConsole;
    this.enableFile = enableFile;
    this.logFile = enableFile ? 'chat-debug.log' : null;
  }

  /**
   * Main logging method with structured format
   * @param {string} event - Event type (connect, message, disconnect, etc.)
   * @param {Object} data - Structured data about the event
   * @param {string} level - Log level (info, warn, error, debug)
   */
  log(event, data = {}, level = 'info') {
    const timestamp = new Date().toISOString();
    const logEntry = {
      timestamp,
      event,
      level,
      pid: process.pid,
      ...data
    };

    // Console output (colored for readability)
    if (this.enableConsole) {
      const color = this._getColorForLevel(level);
      const reset = '\x1b[0m';
      console.log(`${color}[${timestamp}] ${event}: ${JSON.stringify(data)}${reset}`);
    }

    // File output (append)
    if (this.enableFile && this.logFile) {
      const fs = require('fs');
      fs.appendFileSync(this.logFile, JSON.stringify(logEntry) + '\n');
    }
  }
```

```
/**
 * Helper to get console color for log level
 */
_getColorForLevel(level) {
  switch (level) {
    case 'error': return '\x1b[31m'; // Red
    case 'warn': return '\x1b[33m';  // Yellow
    case 'debug': return '\x1b[36m'; // Cyan
    default: return '\x1b[32m';      // Green (info)
  }
}

/**
 * Convenience methods for common events
 */
connectionOpened(connectionId, ip, userAgent) {
  this.log('connection_opened', { connectionId, ip, userAgent });
}

connectionClosed(connectionId, reason, code) {
  this.log('connection_closed', { connectionId, reason, code });
}

messageReceived(connectionId, messageType, rawMessage) {
  this.log('message_received', {
    connectionId,
    messageType,
    messageLength: rawMessage?.length
  }, 'debug');
}

messageBroadcast(roomId, messageId, recipientCount, excludeConnectionId) {
  this.log('message_broadcast', {
    roomId,
    messageId,
    recipientCount,
    excludeConnectionId
  });
}

roomJoined(connectionId, userId, roomId, memberCount) {
  this.log('room_joined', { connectionId, userId, roomId, memberCount });
}
```

```
  roomLeft(connectionId, userId, roomId, memberCount) {
    this.log('room_left', { connectionId, userId, roomId, memberCount });
  }

  /**
   * Dump current server state for debugging
   * @param {ConnectionManager} connectionManager
   * @param {RoomManager} roomManager
   */
  dumpState(connectionManager, roomManager) {
    const state = {
      timestamp: new Date().toISOString(),
      connections: connectionManager?.connections?.size || 0,
      rooms: roomManager?.rooms?.size || 0,
      memoryUsage: process.memoryUsage()
    };
    this.log('state_dump', state, 'debug');
  }
}

// Singleton instance for easy import
module.exports = new DebugLogger(true, false);
```

D. Test WebSocket Client Skeleton

Create a programmatic test client to simulate users and automate testing:

JAVASCRIPT

```javascript
// src/server/debug/test-client.js

/**
 * Programmatic WebSocket client for testing server behavior
 */

const WebSocket = require('ws');

class TestWebSocketClient {
  /**
   * Create a new test client
   * @param {string} url - WebSocket server URL
   */
  constructor(url) {
    this.url = url;
    this.socket = null;
    this.messageQueue = [];  // Stores received messages
    this.messageHandlers = new Map();  // Type -> handler function
    this.connectPromise = null;
  }

  /**
   * Connect to the server with optional authentication token
   * @param {string} token - Optional authentication token
   * @returns {Promise<void>}
   */
  async connect(token = null) {
    return new Promise((resolve, reject) => {
      const urlWithToken = token ? `${this.url}?token=${token}` : this.url;
      this.socket = new WebSocket(urlWithToken);

      this.socket.on('open', () => {
        console.log(`Test client connected to ${this.url}`);
        resolve();
      });

      this.socket.on('message', (rawMessage) => {
        try {
          const message = JSON.parse(rawMessage);
          this.messageQueue.push(message);

          // Call any registered handler for this message type
          const handler = this.messageHandlers.get(message.type);
```

```
        if (handler) {
          handler(message.payload);
        }
      } catch (error) {
        console.error('Failed to parse message:', rawMessage);
      }
    });

    this.socket.on('error', (error) => {
      reject(error);
    });

    this.socket.on('close', (code, reason) => {
      console.log(`Test client disconnected: ${code} - ${reason}`);
    });
  });
}

/**
 * Send a JSON message to the server
 * @param {string} type - Message type
 * @param {Object} payload - Message payload
 */
send(type, payload) {
  if (this.socket && this.socket.readyState === WebSocket.OPEN) {
    const message = { type, payload, timestamp: new Date().toISOString() };
    this.socket.send(JSON.stringify(message));
    return message;
  } else {
    throw new Error('WebSocket not connected');
  }
}

/**
 * Wait for a specific message type with timeout
 * @param {string} messageType - Type to wait for
 * @param {number} timeoutMs - Timeout in milliseconds
 * @returns {Promise<Object>} - The received message
 */
waitForMessage(messageType, timeoutMs = 5000) {
  return new Promise((resolve, reject) => {
```

```
    const timeout = setTimeout(() => {
      reject(new Error(`Timeout waiting for ${messageType}`));
    }, timeoutMs);

    // Check existing messages first
    const existing = this.messageQueue.find(m => m.type === messageType);
    if (existing) {
      clearTimeout(timeout);
      resolve(existing);
      return;
    }

    // Set up handler for future messages
    const handler = (payload) => {
      clearTimeout(timeout);
      resolve({ type: messageType, payload });
    };

    this.messageHandlers.set(messageType, handler);
  });
}

/**
 * Get all received messages of a specific type
 * @param {string} messageType
 * @returns {Array}
 */
getMessagesOfType(messageType) {
  return this.messageQueue.filter(m => m.type === messageType);
}

/**
 * Disconnect from server
 */
disconnect() {
  if (this.socket) {
    this.socket.close();
    this.socket = null;
  }
}

/**
```

```
   * Clear message queue and handlers
   */
  clear() {
    this.messageQueue = [];
    this.messageHandlers.clear();
  }
}

module.exports = TestWebSocketClient;
```

E. State Inspector Implementation

Add this to your ConnectionManager to help debug state issues:

JAVASCRIPT

```javascript
// Add to src/server/connection-manager.js

class ConnectionManager {
  // ... existing code ...

  /**
   * Debug method: Get current connection state for inspection
   * @returns {Object} - Structured state information
   */
  getDebugState() {
    const connections = [];

    for (const [connectionId, session] of this.connections) {
      connections.push({
        connectionId,
        userId: session.userId,
        roomId: session.roomId,
        ip: session.ip,
        connectedAt: session.joinedAt,
        lastActivity: session.lastActivity,
        readyState: session.socket?.readyState
      });
    }

    return {
      totalConnections: this.connections.size,
      connections,
      serverTime: new Date().toISOString(),
      memory: process.memoryUsage()
    };
  }

  /**
   * Debug method: Find connections with issues
   * @returns {Array} - List of problematic connections
   */
  findProblematicConnections() {
    const now = Date.now();
    const problematic = [];

    for (const [connectionId, session] of this.connections) {
      // Check for zombie connections (no activity for > 30 seconds)
```

```
      if (now - session.lastActivity > 30000 && session.socket.readyState === WebSocket.OPEN) {
        problematic.push({
          connectionId,
          issue: 'zombie_connection',
          lastActivity: session.lastActivity,
          idleSeconds: Math.floor((now - session.lastActivity) / 1000)
        });
      }

      // Check for connections without users (shouldn't happen after auth)
      if (!session.userId) {
        problematic.push({
          connectionId,
          issue: 'no_user_id',
          joinedAt: session.joinedAt
        });
      }
    }

    return problematic;
  }
}
```

F. Debugging Test Scenarios

Create automated test scenarios to verify fixes work:

JAVASCRIPT

```javascript
// test/debug-scenarios.js

const TestWebSocketClient = require('../src/server/debug/test-client');

const { wait } = require('./test-utils');


async function testMessageBroadcast() {

  console.log('=== Testing Message Broadcast ===');



  const client1 = new TestWebSocketClient('ws://localhost:8080');

  const client2 = new TestWebSocketClient('ws://localhost:8080');



  try {

    // Connect both clients

    await client1.connect('user1_token');

    await client2.connect('user2_token');



    // Both join same room

    await client1.send('join_room', { roomId: 'test-room' });

    await client2.send('join_room', { roomId: 'test-room' });



    // Wait for join confirmations

    await client1.waitForMessage('room_joined');

    await client2.waitForMessage('room_joined');



    // Client1 sends message

    const sentMessage = client1.send('chat_message', {

      content: 'Hello from client1',

      roomId: 'test-room'

    });



    // Client2 should receive it

    const received = await client2.waitForMessage('chat_message', 2000);



    if (received.payload.content === 'Hello from client1') {

      console.log('✅ Message broadcast test PASSED');

    } else {

      console.log('❌ Message broadcast test FAILED');

    }



  } catch (error) {

    console.log('❌ Message broadcast test FAILED with error:', error.message);

  } finally {
```

```
    client1.disconnect();
    client2.disconnect();
  }
}

async function testConnectionCleanup() {
  console.log('=== Testing Connection Cleanup ===');

  const client = new TestWebSocketClient('ws://localhost:8080');

  try {
    await client.connect('test_token');
    await client.send('join_room', { roomId: 'test-room' });

    // Get initial connection count from server debug endpoint
    const initialCount = await getConnectionCount();

    // Disconnect abruptly
    client.socket.terminate(); // Forceful disconnect

    // Wait for cleanup
    await wait(1000);

    const finalCount = await getConnectionCount();

    if (finalCount < initialCount) {
      console.log('✅ Connection cleanup test PASSED');
    } else {
      console.log('❌ Connection cleanup test FAILED - count:', initialCount, '->', finalCount);
    }

  } catch (error) {
    console.log('❌ Connection cleanup test FAILED with error:', error.message);
  }
}

// Run all debug tests
async function runAllDebugTests() {
  console.log('Starting debug test suite...\n');

  await testMessageBroadcast();
```

```
  console.log();

  await testConnectionCleanup();


  console.log('\nDebug test suite completed');

}


// Helper to get connection count from server

async function getConnectionCount() {

  // Assuming you have a debug HTTP endpoint

  const response = await fetch('http://localhost:8080/debug/connections');

  const data = await response.json();

  return data.totalConnections;

}


// Run if this file is executed directly

if (require.main === module) {

  runAllDebugTests();

}
```

G. Language-Specific Debugging Hints (JavaScript/Node.js)

1. **WebSocket readyState values:**
   - `0` (CONNECTING): Socket has been created but connection not yet open
   - `1` (OPEN): Connection is open and ready to communicate
   - `2` (CLOSING): Connection is in the process of closing
   - `3` (CLOSED): Connection is closed or couldn't be opened
2. **Use Node.js inspector:** Run server with `node --inspect server.js`, then open `chrome://inspect` in Chrome to debug with breakpoints.
3. **Memory leak detection:** Use `node --inspect --trace-gc server.js` and monitor garbage collection in Chrome DevTools Memory tab.
4. **Event loop monitoring:** Use `node --trace-event-categories node.perf server.js` to generate timeline trace.
5. **Handle uncaught exceptions:** Always add:

JAVASCRIPT

```
  process.on('uncaughtException', (error) => {

    console.error('Uncaught exception:', error);

    // Don't exit - WebSocket servers should try to recover

  });


  process.on('unhandledRejection', (reason, promise) => {

    console.error('Unhandled rejection at:', promise, 'reason:', reason);

  });
```

H. Milestone Debugging Checkpoints

After completing each milestone, run these verification tests:

Milestone 1: WebSocket Server Setup

```
Command: node server.js
Test: Open browser console, run: ws = new WebSocket('ws://localhost:8080')
Expected: Server logs "connection_opened", browser shows readyState changes to 1
Debug: If connection fails, check server is listening, firewall, and CORS headers
```

Milestone 2: Message Broadcasting

```
Command: Open two browser tabs to your chat page
Test: Send message from Tab 1
Expected: Message appears in Tab 2 (but not duplicated in Tab 1)
Debug: Use browser Network tab → WS → Messages to see frames
```

Milestone 3: Chat Rooms

```
Command: Join different rooms in different tabs
Test: Send message in Room A (Tab 1), check Room B (Tab 2)
Expected: Message only appears in same room tab
Debug: Check room IDs in message payloads and member sets
```

Milestone 4: Authentication & Persistence

```
Command: Restart server, reload chat page
Test: Send messages, restart server, reload page, rejoin room
Expected: Previous messages loaded from history
Debug: Check database queries, message timestamps, pagination limits
```

Future Extensions

> **Milestone(s):** This section explores enhancements that build upon all four milestones, showing how the foundational architecture can evolve to support more sophisticated features and scale.

The current architecture provides a solid foundation for a real-time chat application that meets the project's learning objectives. However, real-world chat systems often include additional features and face scaling requirements that exceed the initial scope. This section explores potential enhancements, analyzing how the current design accommodates them and what modifications would be necessary. Understanding these extension points helps illustrate how the principles learned—stateful connections, room-based messaging, presence, and persistence—form the basis for more complex systems.

Potential Features and Scaling Paths

The journey from a learning project to a production-ready chat system involves adding user-facing features and addressing scalability limitations. The following enhancements are organized by their impact on the architecture: from simple feature additions that fit within the current component boundaries, to significant architectural changes required for horizontal scaling.

Feature Enhancements Within Current Architecture

Several valuable chat features can be implemented with minimal changes to the core architecture, primarily by extending existing components and adding new message types to the wire protocol.

Direct (Private) Messaging

> **Mental Model:** The Post Office's Registered Mail
>
> While room-based messaging is like broadcasting announcements in a town square, direct messaging is like sending a sealed letter to a specific individual. The postal system (our architecture) already knows how to route messages and track user locations (presence); we just need to add addressing rules that deliver to a single recipient rather than a room's subscriber list.

Direct messaging enables users to send private messages to other users regardless of which rooms they're currently in. This feature requires:

1. **New Message Type:** Add `EVENT_DIRECT_MESSAGE` to the protocol.
2. **Recipient Routing:** Extend the `RoomManager.broadcastToRoom` method or create a new `sendToUser` method that looks up the recipient's current `ClientSession` via their `userId`.
3. **Presence Check:** Verify the recipient is online before sending, with optional offline message queuing.
4. **Privacy Controls:** Allow users to block messages from specific users.

The current data model already supports this: `ClientSession` links a `userId` to a specific connection. The main architectural addition is a user-to-connection lookup index in the `ConnectionManager` or `RoomManager`.

Component	Changes Required	Data Flow Impact
Wire Protocol	Add `EVENT_DIRECT_MESSAGE` type with `recipientId` field	New message flow: Client → Server → Specific Client
ConnectionManager	Maintain `userId` → `connectionId` lookup map	Enables direct routing without room context
RoomManager	Add `sendToUser(userId, message)` method	Uses ConnectionManager's lookup to deliver
Client Application	Add UI for user list and private chat windows	New subscription to direct message events

Message Reactions and Threads

> **Mental Model:** Marginal Notes and Conversation Threads
>
> Reactions are like scribbling an emoji in the margin of a printed chat transcript—a lightweight annotation attached to a specific message. Threads are like starting a sidebar conversation linked to a particular comment in a meeting minutes document.

Reactions (e.g., 👍, ❤️) allow users to respond to specific messages without creating new chat messages. Threads enable nested conversations about specific messages. Both features require:

1. **Message References:** Extend `ChatMessage` with optional `parentMessageId` and `reactions` field (a map of emoji→userIds).
2. **New Event Types:** `EVENT_REACTION_ADDED`, `EVENT_THREAD_REPLY`.
3. **Persistence Updates:** Modify `storeMessage` to handle parent relationships and update reactions atomically.
4. **Efficient Broadcast:** When a reaction is added, broadcast only the reaction delta, not the entire message history.

The current `ChatMessage` structure can be extended with additional fields. The `RoomManager` would need logic to validate that referenced messages exist in the same room.

Read Receipts

> **Mental Model:** Certified Mail with Return Receipt
>
> Read receipts provide sender confirmation that their message was seen by the recipient, similar to certified mail that requires a signature upon delivery. The system must track which users have "seen" each message.

Read receipts show when messages have been viewed by recipients. Implementation requires:

1. **Client-Side Visibility Tracking:** The client must detect when a message becomes visible in the viewport and send a `EVENT_MESSAGE_READ` event.
2. **Server-Side State Management:** The server must track which users have read which messages (a `messageId` → `Set<userId>` mapping).
3. **Efficient Updates:** When a user reads multiple messages (e.g., scrolling through history), batch updates are preferable to individual events.
4. **Broadcast Updates:** Notify the message sender when their messages are read (optional).

This feature adds significant state management complexity. A separate `ReadReceiptService` component might be warranted to avoid bloating the `RoomManager`.

Rich Media and File Sharing

> **Mental Model:** The Package Delivery Service
>
> While text messages are like postcards (small, simple), file sharing is like shipping packages. We need a separate handling process: receiving the large payload, storing it in a warehouse (object storage), and providing a delivery receipt (URL) that recipients can use to retrieve it.

Allowing users to share images, documents, and other files requires:

1. **Separate Upload Pathway:** Large files shouldn't go through WebSocket binary messages. Implement HTTP POST endpoints for file upload.
2. **Object Storage Integration:** Store files in cloud storage (S3, GCS) or a local filesystem with access controls.
3. **Message with Metadata:** Send chat messages containing file metadata (URL, filename, size, MIME type) via WebSocket.
4. **Security Considerations:** Validate file types, scan for malware, implement access tokens for private files.

The current architecture's separation between WebSocket messaging and HTTP services (for authentication) provides a pattern for adding additional HTTP endpoints.

Bot Integrations and Webhooks

> **Mental Model:** The Conference Call Interpreter
>
> Bots are like interpreters joining a conference call—they listen to conversations, can be addressed directly with commands, and can inject their own messages. Webhooks are like setting up a microphone in the room that streams the conversation to external services.

Bots are automated users that respond to commands or patterns. Webhooks allow external services to receive room events. Both require:

1. **Programmatic Client API:** A simplified client SDK for bots to connect and interact.
2. **Command Parsing:** Middleware to detect bot commands (e.g., `/giphy cats`).
3. **Rate Limiting:** Prevent bots from spamming rooms.
4. **Webhook Delivery:** HTTP callback system to POST room events to external URLs.

The current `RoomManager.broadcastToRoom` method already delivers messages to all connections; bots would simply be additional connections with special processing logic.

Advanced Presence States

> **Mental Model:** The Office Status Light
>
> Beyond simple online/offline, users can have statuses like "away," "do not disturb," or "in a meeting"—similar to an office door having a light that shows availability for interruptions.

Extend the basic presence system to show user availability states:

1. **State Transitions:** Define states (`online`, `away`, `busy`, `offline`) with automatic transitions (e.g., `away` after keyboard inactivity).
2. **Client Heartbeat Enhancement:** Extend the heartbeat mechanism to include user activity state.
3. **Status Broadcast:** Update `EVENT_USER_PRESENCE` to include detailed status and optional custom message.
4. **Privacy Controls:** Allow users to hide detailed presence from certain users.

The `ClientSession` already has `lastActivity`; this can be extended with `presenceState` and `statusMessage` fields.

Scaling Beyond a Single Server

The current architecture assumes a single Node.js process handling all connections. While this can support thousands of concurrent users (depending on message volume), true horizontal scaling requires significant architectural changes.

Decision: Scaling WebSocket Connections Across Multiple Servers

- **Context:** The single-server architecture faces limits on concurrent connections (memory, CPU, file descriptor limits). To support tens or hundreds of thousands of users, connections must be distributed across multiple servers.
- **Options Considered:**
  1. **Load Balancer with IP Hashing:** A TCP load balancer (L4) distributes connections based on client IP, keeping each user's connection on the same server.
  2. **WebSocket-Aware Load Balancer:** A layer 7 load balancer that understands WebSocket protocol and can route based on cookies or custom headers.
  3. **Connection Routing Service:** A dedicated service that assigns clients to specific backend servers and handles re-routing.
- **Decision:** Use a WebSocket-aware load balancer (option 2) for simplicity, combined with a shared persistence layer.
- **Rationale:** Layer 7 load balancers (like Nginx with WebSocket support) are mature, require minimal application changes, and can handle SSL termination. IP hashing (option 1) breaks when users change networks or use load-balanced mobile networks.
- **Consequences:** Requires all servers to have access to shared room state and message history. The in-memory room state becomes invalid; we must move to external storage.

Option	Pros	Cons	Viability
IP Hashing Load Balancer	Simple, works at TCP layer, minimal configuration	Same user from different IPs may connect to different servers, breaking state	Poor - breaks user experience
WebSocket-Aware Load Balancer	Can use cookies/sessions for sticky routing, handles WebSocket upgrade	More complex configuration, requires session affinity	**Good** - balances simplicity and functionality
Connection Routing Service	Maximum flexibility, can implement custom routing logic	Additional service to build and maintain, new failure point	Overkill for most applications

Shared State with Redis or Database When moving to multiple servers, the in-memory room and presence state must be externalized to a shared data store. Redis is ideal for this due to its low-latency data structures and publish/subscribe capabilities.

Current Component	Multi-Server Adaptation	Redis Data Structures
ConnectionManager.activeConnections	Each server maintains only its local connections	Not shared; ephemeral connection state remains local
RoomManager.rooms (member lists)	Store in Redis Sorted Sets or Hashes	`room:members:{roomId}` → Set of `userId` s
User Presence Status	Store in Redis with TTL	`user:presence:{userId}` → Hash with status, lastSeen
Typing Indicators	Store in Redis with short TTL	`room:typing:{roomId}` → Hash of `userId` → timestamp
Message Routing	Use Redis Pub/Sub for cross-server broadcasts	Publish to channel `room:{roomId}`; all servers subscribe

Stateless WebSocket Servers with External Message Queue For maximum scalability, the WebSocket servers can become stateless connection handlers that delegate all business logic to external services via message queues.

> **Mental Model:** The Airport Gate Agents vs. Air Traffic Control
>
> In this model, WebSocket servers are like gate agents—they manage the direct interaction with passengers (clients) but don't make routing decisions. A central "air traffic control" (message queue + worker services) coordinates all flights (messages) between gates.

1. **WebSocket Servers** only handle connection lifecycle and raw message I/O.
2. **All business logic** (room joins, message validation, presence updates) moves to separate worker services.
3. **A message queue** (like RabbitMQ, Kafka, or Redis Streams) routes events between components.
4. **Connection routing layer** tracks which server hosts each user's connection.

This is a significant architectural shift that introduces complexity but enables independent scaling of connection handling versus business logic processing.

ADR: When to Consider Multi-Server Architecture

- **Context:** Determining when the single-server architecture becomes inadequate and requires the complexity of distributed systems.
- **Options Considered:**
  1. **Scale from Day One:** Implement multi-server architecture even for the learning project.
  2. **Scale When Needed:** Start with single-server, refactor when hitting limits.
  3. **Hybrid Approach:** Design with extension points for distribution but implement single-server first.
- **Decision:** Hybrid approach (option 3)—design clean interfaces between components so they can be distributed later without rewriting application logic.
- **Rationale:** Premature optimization distracts from learning core concepts. However, anticipating future scaling informs better interface design today (e.g., abstracting storage behind interfaces).
- **Consequences:** Some refactoring will still be needed when scaling, but the core application logic remains largely unchanged.

Security and Moderation Enhancements

Production chat systems require robust security and moderation features that go beyond basic authentication.

End-to-End Encryption

> **Mental Model:** The Diplomatic Pouch
>
> End-to-end encryption ensures only the intended recipients can read messages, like diplomatic documents sealed in a tamper-proof pouch that only the sender and recipient have keys to open.

Implementing true end-to-end encryption (where the server cannot decrypt messages) requires:

1. **Key Exchange:** Clients must exchange public keys (via the server as an untrusted channel).
2. **Client-Side Encryption:** Messages are encrypted/decrypted in the client application using WebCrypto API.
3. **Server as Message Relay:** The server stores and forwards encrypted blobs without understanding content.
4. **Key Management:** Handle key loss, device addition, and group chats (which require more complex encryption schemes).

This fundamentally changes the server's role from message processor to blind router and significantly increases client complexity.

Content Moderation and Filtering

> **Mental Model:** The Editor's Red Pen
>
> Automated moderation scans messages for inappropriate content before or after they're sent, similar to an editor reviewing articles before publication.

Moderation features include:

1. **Automated Keyword Filtering:** Scan messages for banned words/phrases.
2. **Image Analysis:** Use AI services to detect inappropriate images.
3. **Report System:** Allow users to report offensive messages.
4. **Moderator Dashboard:** Interface for human moderators to review reports and take action.

Implementation requires message processing pipeline extensions—potentially asynchronous processing where messages are queued for moderation review before being broadcast.

Rate Limiting and Anti-Spam

> **Mental Model:** The Conversation Pacekeeper
>
> Rate limiting prevents users from overwhelming the system with too many messages too quickly, like a moderator ensuring everyone gets a turn to speak.

Protect the system from abuse by:

1. **Connection-Level Rate Limiting:** Limit message frequency per connection.
2. **User-Level Rate Limiting:** Track message counts per user across all connections.
3. **Content-Based Rules:** Detect and block spam patterns (repeated messages, excessive mentions).
4. **Graduated Response:** Start with warnings, then temporary mutes, then bans.

The `ConnectionManager` would need to track message timestamps per connection/user and implement sliding window algorithms.

Performance Optimizations

As user count and message volume grow, performance optimizations become necessary.

Message Compression WebSocket messages can be compressed using permessage-deflate extension (built into most WebSocket libraries). For text-heavy chat, compression can reduce bandwidth by 70-90%.

Client-Side Message Caching and Delta Updates Instead of sending full message history each time a user joins a room, clients can cache messages locally and request only updates since their last view. This requires:

1. **Client Storage:** Use IndexedDB or localStorage to cache messages per room.
2. **Sync Tokens:** Server provides synchronization tokens representing state versions.
3. **Delta Calculations:** Server computes differences between client cache and current state.

Efficient Presence Broadcasts Broadcasting presence updates for large rooms (1000+ users) can create significant load. Optimizations include:

1. **Throttling Updates:** Batch presence changes and send less frequently.
2. **Lazy Loading:** Only load presence for users currently visible in the client viewport.
3. **Presence Channels:** Separate presence updates from chat messages, allowing clients to opt-in based on interest.

Implementation Guidance

The following guidance provides practical steps for implementing some of the simpler extensions while maintaining the current architecture.

Technology Recommendations for Extensions

Feature	Simple Option (Learning Focus)	Production-Ready Option
Direct Messaging	Extend `RoomManager` with user lookup map	Implement with dedicated messaging service
File Sharing	Local filesystem storage with size limits	Cloud object storage (S3) with CDN
Bot Framework	Simple command parser in `RoomManager`	Bot SDK with middleware pipeline
Multi-Server Scaling	Sticky sessions with Redis for shared state	Full microservices with message queue
End-to-End Encryption	Basic symmetric encryption demo	Proper key management with Signal Protocol
Moderation	Keyword filter middleware	AI content moderation API integration

File Structure for Extensions

```
chat-project/
├── server/
│   ├── src/
│   │   ├── core/                   # Existing core components
│   │   │   ├── ConnectionManager.ts
│   │   │   ├── RoomManager.ts
│   │   │   └── AuthService.ts
│   │   ├── extensions/             # New extensions directory
│   │   │   ├── DirectMessaging.ts   # Direct message handler
│   │   │   ├── FileService.ts       # File upload/download
│   │   │   ├── BotFramework.ts      # Bot command processing
│   │   │   ├── Moderation.ts        # Content filtering
│   │   │   └── RateLimiter.ts       # Rate limiting middleware
│   │   ├── shared/                 # Shared utilities
│   │   │   ├── redis.ts            # Redis client for shared state
│   │   │   └── encryption.ts       # Encryption helpers
│   │   └── main.ts                 # Updated to load extensions
│   └── uploads/                    # Local file storage (if used)
└── client/
    └── src/
        ├── extensions/             # Client-side extension code
        │   ├── DirectMessageUI.tsx  # Private chat UI component
        │   └── FileUploader.tsx     # File upload component
        └── crypto/                 # Client-side encryption (if needed)
```

Starter Code for Direct Messaging Extension

```javascript
// server/src/extensions/DirectMessaging.ts

import { ConnectionManager } from '../core/ConnectionManager';

import { RoomManager } from '../core/RoomManager';

import { createServerMessage } from '../shared/protocol';


// New event type constant

const EVENT_DIRECT_MESSAGE = 'direct_message';

const EVENT_DIRECT_MESSAGE_DELIVERED = 'direct_message_delivered';

const EVENT_DIRECT_MESSAGE_ERROR = 'direct_message_error';


/**

 * Direct Messaging Extension

 *

 * This extension adds private messaging between users.

 * It maintains a map of userId -> connectionId for online users

 * and routes direct messages to the appropriate recipient.

 */

export class DirectMessaging {

  private userConnections = new Map(); // userId -> connectionId

  private connectionUsers = new Map(); // connectionId -> userId


  constructor(

    private connectionManager: ConnectionManager,

    private roomManager: RoomManager

  ) {

    // Listen for user authentication events

    this.setupEventHandlers();

  }


  /**

   * Register a user's current connection

   */

  registerUserConnection(userId: string, connectionId: string): void {

    // TODO 1: Update userConnections map with userId -> connectionId

    // TODO 2: Update connectionUsers map with connectionId -> userId

    // TODO 3: Clean up old mapping if user had previous connection

  }


  /**

   * Remove a user's connection mapping
```

```
 */
unregisterUserConnection(connectionId: string): void {
  // TODO 1: Get userId from connectionUsers map using connectionId
  // TODO 2: Remove entry from connectionUsers map
  // TODO 3: Remove entry from userConnections map if it maps to this connectionId
}

/**
 * Handle incoming direct message from a client
 */
async handleDirectMessage(senderConnectionId: string, messagePayload: any): Promise<void> {
  // TODO 1: Extract recipientId and content from messagePayload
  // TODO 2: Validate recipientId exists and is not the sender
  // TODO 3: Look up recipient's connectionId from userConnections map
  // TODO 4: If recipient is online, forward message using connectionManager.sendToClient
  // TODO 5: Send delivery confirmation to sender
  // TODO 6: If recipient is offline, store message for later delivery (optional)
  // TODO 7: Handle errors (invalid recipient, message too long, etc.)
}

/**
 * Check if a user is currently online
 */
isUserOnline(userId: string): boolean {
  // TODO: Check if userId exists in userConnections map
  return this.userConnections.has(userId);
}

/**
 * Get the connection ID for an online user
 */
getConnectionForUser(userId: string): string | undefined {
  // TODO: Return connectionId from userConnections map
  return this.userConnections.get(userId);
}

private setupEventHandlers(): void {
  // TODO: Listen for user authentication events from RoomManager
  // TODO: Listen for connection disconnect events from ConnectionManager
  // TODO: Register message handler for EVENT_DIRECT_MESSAGE
```

```
  }
}
```

Starter Code for File Service Extension

```javascript
// server/src/extensions/FileService.ts

import fs from 'fs/promises';

import path from 'path';

import { v4 as uuidv4 } from 'uuid';


// Constants for file handling

const MAX_FILE_SIZE = 10 * 1024 * 1024; // 10MB

const ALLOWED_MIME_TYPES = [

  'image/jpeg',

  'image/png',

  'image/gif',

  'application/pdf',

  'text/plain'

];

const UPLOAD_DIR = './uploads';


/**

 * File Service for handling file uploads and downloads

 *

 * This service provides HTTP endpoints for file upload

 * and manages file metadata for inclusion in chat messages.

 */

export class FileService {

  private fileMetadata = new Map(); // fileId -> { filename, mimeType, size, uploader, uploadDate }


  constructor() {

    this.ensureUploadDirectory();

  }


  /**

   * Ensure upload directory exists

   */

  private async ensureUploadDirectory(): Promise<void> {

    try {

      await fs.access(UPLOAD_DIR);

    } catch {

      await fs.mkdir(UPLOAD_DIR, { recursive: true });

    }

  }
```

```
  /**
   * Handle file upload from HTTP request
   */
  async handleUpload(fileData: Buffer, filename: string, mimeType: string, uploaderId: string): Promise<{ fileId: string, url: string }> {
    // TODO 1: Validate file size against MAX_FILE_SIZE
    // TODO 2: Validate mimeType against ALLOWED_MIME_TYPES
    // TODO 3: Generate unique fileId using uuidv4()
    // TODO 4: Create safe filename (prevent path traversal)
    // TODO 5: Write file to disk in UPLOAD_DIR
    // TODO 6: Store metadata in fileMetadata map
    // TODO 7: Return fileId and URL for client access
  }

  /**
   * Get file metadata by ID
   */
  getFileMetadata(fileId: string): any {
    // TODO: Return metadata from fileMetadata map
    return this.fileMetadata.get(fileId);
  }

  /**
   * Serve file for download
   */
  async serveFile(fileId: string): Promise<{ data: Buffer, metadata: any }> {
    // TODO 1: Look up metadata by fileId
    // TODO 2: Read file from disk
    // TODO 3: Return file data and metadata
  }

  /**
   * Clean up old files (cron job)
   */
  async cleanupOldFiles(maxAgeDays: number = 30): Promise<void> {
    // TODO 1: Iterate through fileMetadata map
    // TODO 2: Check uploadDate for each file
    // TODO 3: Delete files older than maxAgeDays
    // TODO 4: Remove entries from fileMetadata map
  }
}
```

Language-Specific Hints for JavaScript/TypeScript

- **Direct Messaging:** Use `Map` for `userId → connectionId` lookups for O(1) access. Remember to clean up mappings when connections close to prevent memory leaks.
- **File Uploads:** For production, use streams (`fs.createWriteStream`) instead of writing entire buffers to handle large files efficiently.
- **Redis Integration:** Use the `ioredis` library for Redis connections with connection pooling and automatic reconnection.
- **Rate Limiting:** Implement sliding window algorithm using Redis sorted sets for distributed rate limiting across multiple servers.
- **Bot Framework:** Create a middleware pipeline pattern where messages pass through bot handlers sequentially until one consumes the message.

Milestone Extension Checkpoints

Direct Messaging Implementation Check:

1. Start the server and connect two authenticated clients (Alice and Bob).
2. Alice sends a direct message to Bob using the new message type.
3. Verify Bob receives the message immediately.
4. Verify Alice receives a delivery confirmation.
5. Test offline case: disconnect Bob, send message from Alice, then reconnect Bob—message should not be delivered unless you implemented offline queueing.

File Sharing Implementation Check:

1. Start server with file upload endpoint.
2. Use curl or Postman to POST a file to `/api/upload` .
3. Verify server returns a `fileId` and URL.
4. Send a chat message containing the file metadata.
5. Other clients should see the file message and be able to download it via the provided URL.

Multi-Server Scaling Check (Advanced):

1. Start two server instances on different ports.
2. Configure a load balancer (nginx) with sticky sessions.
3. Connect multiple clients, ensuring they distribute across servers.
4. Using Redis, verify that room membership and presence updates propagate across servers.
5. Send a message in a room with users connected to different servers—all should receive it.

Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
Direct messages not delivered	User connection mapping not updated	Check `userConnections` map after login	Ensure `registerUserConnection` is called on authentication
File uploads failing silently	Upload directory permissions	Check server logs for EACCES errors	Ensure upload directory exists and is writable
Presence not syncing across servers	Redis connection issues	Check Redis connectivity and Pub/Sub subscriptions	Verify Redis URL and ensure all servers subscribe to same channels
Rate limiting blocking legitimate users	Algorithm too aggressive	Log rate limit counters and timestamps	Adjust time window or increase limit thresholds
Bots not responding to commands	Command prefix mismatch	Log raw messages entering bot handler	Ensure command parsing logic matches client sending format

Glossary

> **Milestone(s):** All milestones (1-4). This glossary defines key terms and concepts used throughout the design document, providing a common vocabulary for understanding the real-time chat application architecture.

Terms and Definitions

Term	Definition
Authentication	The process of verifying a user's identity before granting access to the system, typically by validating credentials (username and password) during the WebSocket connection upgrade or through a custom protocol message. In this system, authentication occurs during the HTTP upgrade to WebSocket, creating a secure link between a user identity and their WebSocket connection.
bcrypt	A password hashing algorithm specifically designed to be computationally expensive and resistant to brute-force attacks through built-in salting and configurable work factors. The system uses bcrypt with `saltRounds: 10` to securely hash user passwords before storage, ensuring plaintext passwords are never persisted in the database.
Broadcast	The action of sending a message to all connected clients in a specific group or room, excluding the original sender by default. Implemented through `RoomManager.broadcastToRoom()`, this pattern enables one-to-many communication essential for chat rooms where a single message must reach multiple recipients simultaneously.
ChatMessage	The core data structure representing an individual chat message, containing fields for `sender` (username), `content` (text), `timestamp` (ISO string), `roomId` (target room), and `id` (unique identifier). This structure flows from client to server, gets enriched with metadata, stored in persistence, and broadcast to room members.
ClientSession	A server-side data structure tracking an authenticated user's active WebSocket connection, containing fields like `connectionId` (unique socket identifier), `socket` (raw WebSocket object), `userId` (linked user), `ip` (client IP address), `userAgent` (browser info), `joinedAt` (connection timestamp), `lastActivity` (for timeout detection), and `roomId` (current room). This object represents the stateful link between user identity and network connection.
Clock Skew	The difference in system time between client and server machines, which can cause inconsistencies in message timestamps and presence indicators. The system mitigates this by having the server assign all timestamps (`ChatMessage.timestamp`) using its own clock, ensuring consistent ordering and time-based operations across all clients.
Connection Lifecycle	The complete sequence of states a WebSocket connection passes through: initial HTTP handshake (upgrade), connected (open), messaging (data exchange), and disconnection (close/error). The `ConnectionManager` tracks this lifecycle through events (connect, message, disconnect) and maintains corresponding state in `ClientSession` objects.
ConnectionManager	The server component responsible for accepting WebSocket upgrade requests, managing active connections, routing messages to appropriate handlers, and cleaning up disconnected clients. This component acts as the system's entry point for all real-time traffic and maintains the `ClientSession` Map tracking all active connections.
Debouncing	A programming technique that delays execution of a function until after a specified wait time has elapsed since the last time it was invoked, used in this system to prevent excessive `EVENT_USER_TYPING` notifications. When a user types, the client waits for a pause (e.g., 500ms) before sending the typing indicator, and sends a "stopped typing" message after a timeout.
DebugLogger	A utility component with fields `enableConsole` , `enableFile` , and `logFile` that provides structured logging for system events at different severity levels. Used throughout the server to record connection events, message handling errors, and system state for troubleshooting, implementing the non-functional goal of debuggability.
Defensive Programming	A coding philosophy that anticipates and gracefully handles potential errors before they cause system failures. In this chat system, this manifests as validating all incoming messages (`parseAndValidateWebSocketMessage`), checking socket `readyState` before sending, implementing comprehensive error handlers, and using try-catch blocks around all message processing.
Direct Messaging	Private, one-to-one communication between individual users outside of chat rooms, implemented as a future extension. The `DirectMessaging` component would manage user-to-connection mappings and route private messages using the `EVENT_DIRECT_MESSAGE` event type while maintaining the same underlying WebSocket connections.
End-to-End Encryption	An encryption model where messages are encrypted on the sender's device and only decrypted on the recipient's device, preventing even the server from reading message contents. This represents a potential future extension beyond the current system's scope, which currently encrypts data only in transit (TLS) and at rest (database).
Event-Driven Architecture	A software design pattern where system components emit and respond to events (messages signaling state changes) rather than using direct method calls. The chat system employs this pattern extensively: the WebSocket server emits connection events, clients send message events (`EVENT_CHAT_MESSAGE` , `EVENT_USER_TYPING`), and the server broadcasts notification events to clients.
Exponential Backoff	A retry strategy where the delay between retry attempts increases exponentially (e.g., 1s, 2s, 4s, 8s), used by clients when attempting to reconnect after losing WebSocket connection. This prevents overwhelming the server with reconnection attempts during outages while ensuring eventual reconnection when the server becomes available.
Graceful Degradation	The property of a system to continue operating with reduced functionality when parts fail, rather than failing completely. In this chat application, if the persistence layer fails, the system continues to allow real-time messaging (using in-memory state) but displays an error when users try to load message history, maintaining core functionality while indicating the degraded state.
Heartbeat	Periodic ping/pong messages exchanged between client and server to detect if a connection is still alive, even if no application messages are being sent. Implemented as a timer in the `ConnectionManager` that sends WebSocket protocol ping frames and disconnects clients that don't

Term	Definition
	respond with pong frames within a timeout period, preventing "zombie connections."
Horizontal Scaling	Adding more servers to handle increased load, as opposed to vertical scaling (upgrading hardware on a single server). While not implemented in the current design (a non-goal), the architecture could be extended for horizontal scaling using Redis Pub/Sub for cross-server communication and sticky sessions for load balancing.
Integration Tests	Tests that verify multiple system components work together correctly, such as testing the full flow from client WebSocket connection through message broadcasting to other clients. In this project, integration tests would verify that messages sent to a room are received by all members, and that user join notifications work properly.
JWT (JSON Web Token)	A compact, URL-safe token format for representing claims between two parties, often used for authentication. While the current system uses session tokens stored server-side, JWTs could be implemented as a future extension to enable stateless authentication, containing encrypted user identity and expiration information.
Long Polling	A technique where the client sends a request to the server that remains open until the server has new data to send, at which point the server responds and the client immediately sends another request. This is an alternative to WebSockets for real-time communication but introduces higher latency and server overhead compared to the persistent WebSocket connections used in this system.
Message Broker	A component that routes messages to appropriate recipients based on rules or subscriptions. In this system, the `RoomManager` acts as a message broker, receiving messages from the `ConnectionManager` and delivering them to all members of the specified room, implementing the publish-subscribe pattern for room-based communication.
Message Queue	In the context of testing, a collection (array) of received messages in the `TestWebSocketClient` that allows test code to inspect messages sent by the server. The client's `messageQueue` field stores all incoming messages, and helper methods like `getMessagesOfType()` filter this queue for specific message types during test verification.
Middleware Pipeline	A pattern where incoming messages pass through a series of processing functions (middleware) before reaching the main handler. While not explicitly implemented in the current design, the architecture could be extended with middleware for logging, rate limiting, message transformation, or authentication validation before messages reach the main `ConnectionManager.handleMessage()` method.
Pagination	The technique of retrieving data in manageable chunks (pages) rather than loading entire datasets at once. Implemented in `getMessagesForRoom()` with `limit` and `offset` parameters, this prevents overwhelming clients and servers when loading message history from rooms with thousands of messages, a critical consideration for the non-functional goal of performance.
Password Hashing	The cryptographic transformation of passwords into irreversible hash values before storage, preventing exposure of plaintext passwords even if the database is compromised. The system uses bcrypt hashing in the `AuthService.registerUser()` and `AuthService.loginUser()` methods, ensuring passwords are never stored or transmitted in plain text.
Polling	A technique where the client repeatedly requests updates from the server at regular intervals (e.g., every 2 seconds). This naive approach to real-time communication is inefficient compared to WebSockets, as it creates constant HTTP request overhead even when no new data is available, and introduces latency equal to the polling interval.
Presence	Real-time status information about users, including whether they are online/offline, which room they're in, and whether they're currently typing. The `RoomManager` tracks presence through room membership (`Room.memberIds`) and typing status via `setUserTyping()` , broadcasting updates to room members when presence states change.
Rate Limiting	Controlling the frequency of messages a user can send to prevent abuse and protect server resources. While not implemented in the basic system, this could be added as a future extension using algorithms like the sliding window algorithm to count requests over a moving time window and reject excessive messages.
Redis Pub/Sub	Redis Publish/Subscribe pattern that enables cross-server communication through a shared message bus, allowing multiple chat servers to coordinate room membership and message broadcasting in a horizontally scaled deployment. This represents an advanced extension path beyond the current single-server, in-memory design.
Room	A data structure representing a chat channel or conversation space, containing fields `id` (unique identifier), `name` (human-readable name), `createdAt` (timestamp), and `memberIds` (Set of user IDs currently in the room). Rooms enable segmented conversations where messages are only delivered to members of that specific room.
Room Manager	The server component that manages chat rooms, membership, message broadcasting within rooms, and presence tracking. This component implements methods like `joinRoom()` , `leaveRoom()` , `broadcastToRoom()` , and `getRoomList()` , forming the core business logic for Milestones 2 and 3.
Server-Sent Events (SSE)	An HTTP-based technology for server-to-client event streaming over a single, long-lived connection where the server can push events to the client but the client cannot send messages back (unidirectional). While simpler than WebSockets, SSE is insufficient for this chat application's requirement for bidirectional communication (client-to-server messaging).
Session	Temporary authentication state for a user connection, represented server-side by `ClientSession` objects and client-side by session tokens. Sessions have a Time-To-Live (`sessionTTL` of 24 hours) after which they expire, requiring re-authentication. The `AuthService` creates, validates, and invalidates sessions.

Term	Definition
Sliding Window Algorithm	A rate limiting algorithm that counts requests over a moving time window (e.g., last 60 seconds) rather than fixed intervals, providing more accurate and smooth limiting. This represents a potential implementation approach for adding rate limiting as a future extension to prevent spam and abuse.
Stateful Message Router	A server component that maintains active connections and routes messages between them based on application state. The `ConnectionManager` combined with the `RoomManager` forms a stateful message router, as it must know which connections belong to which users, which rooms those users are in, and route messages accordingly.
Sticky Sessions	A load balancing technique that routes all requests from a particular user to the same server, necessary for maintaining WebSocket connection state in horizontally scaled deployments. This would be required if extending the system to multiple servers, ensuring a user's WebSocket connection and session state remain on a single server.
TestWebSocketClient	A testing utility that simulates a chat client for automated testing, containing fields like `url`, `socket`, `messageQueue`, `messageHandlers`, and `connectPromise`. This client connects to the server, sends messages, and provides methods like `waitForMessage()` to verify server responses in integration tests.
TTL (Time-To-Live)	The duration after which data expires and should be cleaned up. In this system, sessions have a TTL of 24 hours (stored in `sessionTTL`), and the `cleanupExpiredSessions()` method periodically removes expired sessions. Message history could also implement TTL for automatic cleanup of old messages.
Unicode Normalization	The process of converting text to a standard form for consistent processing, ensuring that visually identical strings (e.g., "café" composed differently) are treated as identical. While not explicitly implemented, this would be important for proper handling of usernames and message content containing international characters.
Unit Tests	Tests that verify individual functions in isolation, such as testing `parseAndValidateWebSocketMessage()` without requiring a running server or network connections. These tests focus on specific edge cases and error conditions for single components or functions.
User	A data structure representing an authenticated chat user, containing `username` (unique identifier), `passwordHash` (bcrypt-hashed password), and `createdAt` (registration timestamp). Users are created during registration and authenticated during login, with their identity linked to WebSocket connections via `ClientSession` objects.
WebSocket	A communication protocol providing full-duplex (bidirectional) communication channels over a single TCP connection, enabling persistent, low-latency communication between client and server. This protocol forms the foundation of the entire chat system, replacing the request-response model of HTTP with continuous two-way communication.
WebSocketMessage	The standardized message format for all client-server communication, containing fields `type` (event type like `EVENT_CHAT_MESSAGE`), `payload` (event-specific data), and optional `timestamp`. All messages, whether from client to server or server to client, follow this envelope format for consistent parsing and routing.
Zombie Connection	A WebSocket connection that appears open in the server's tracking but is actually dead or unresponsive, often due to network failures where the TCP connection closed without proper WebSocket close handshake. The system prevents zombie connections through heartbeat (ping/pong) mechanisms that detect and clean up dead connections.