Real-Time Chat Application

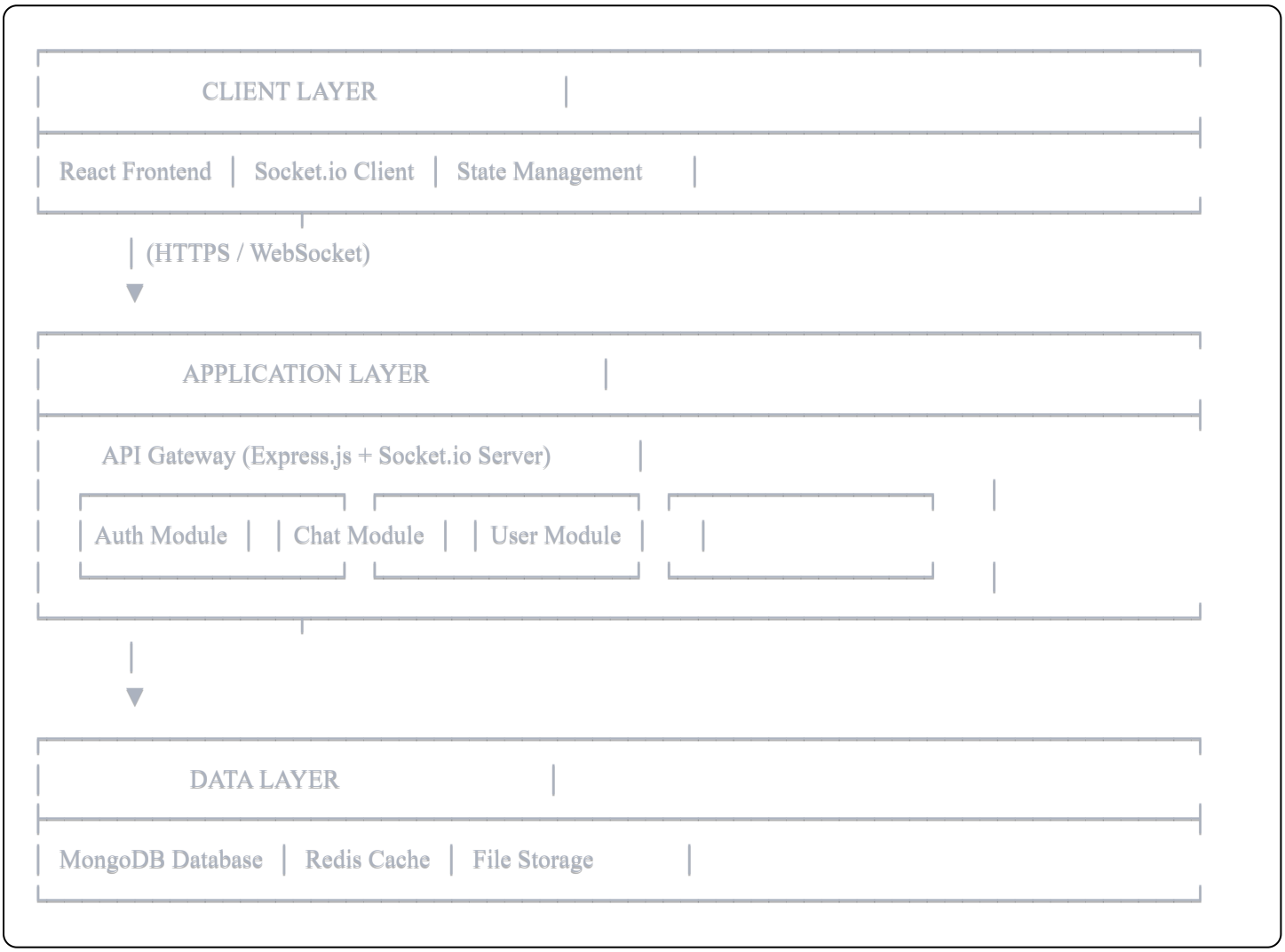
System Design Document

---

1. System Architecture Overview

High-Level Architecture

The application follows a three-tier architecture designed for scalability, performance, and real-time communication capabilities.



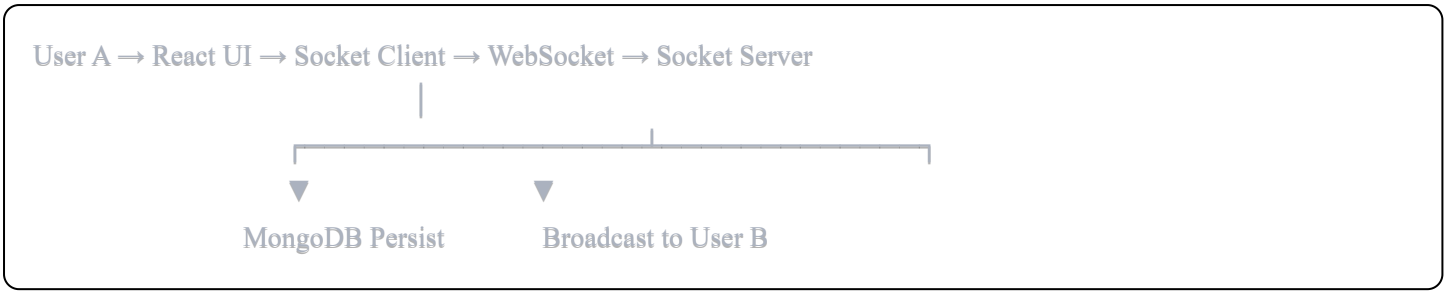
Technology Stack

Frontend: React.js provides component-based architecture with Socket.io-client enabling real-time bidirectional communication. React Router handles navigation while Context API manages application state.

Backend: Node.js with Express.js offers non-blocking I/O perfect for concurrent connections. Socket.io manages real-time events through WebSocket protocol. MongoDB provides flexible NoSQL storage for messages and user data. JWT enables stateless authentication supporting horizontal scaling.

Communication Architecture

Message Transmission Flow:



The system uses a hybrid communication approach. Real-time events (messaging, typing, presence) utilize WebSocket connections for instant delivery. Traditional operations (authentication, message history, search) use REST API with HTTP protocol. This design leverages WebSocket speed while maintaining REST reliability for non-real-time operations.

---

2. Database Schema & Data Management

Collections Structure

Users Collection stores account information including unique identifier, name, email (indexed and unique), hashed password, avatar URL, online status flag, last seen timestamp, and creation metadata.

Conversations Collection manages chat sessions with unique identifier, array of participant IDs (indexed), conversation type (direct or group), optional name for groups, embedded last message preview (content, sender, timestamp), and creation metadata.

Messages Collection contains individual messages with unique identifier, conversation reference (indexed), sender ID, message content, type indicator (text/image/file), delivery status (sent/delivered/read), array of users who read the message, and timestamp information.

Indexing Strategy

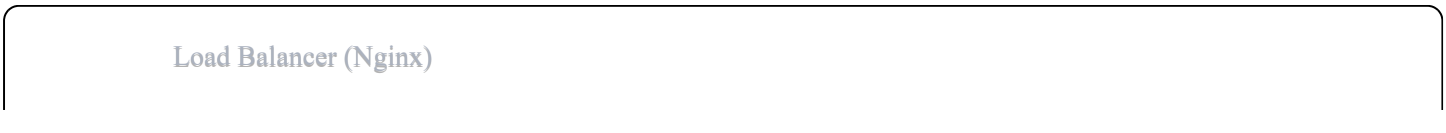
Performance optimization employs strategic indexing: single field index on user email enables fast authentication lookups; compound index on conversation ID and timestamp optimizes message retrieval; array index on conversation participants speeds up filtering; text index on user names supports search functionality.

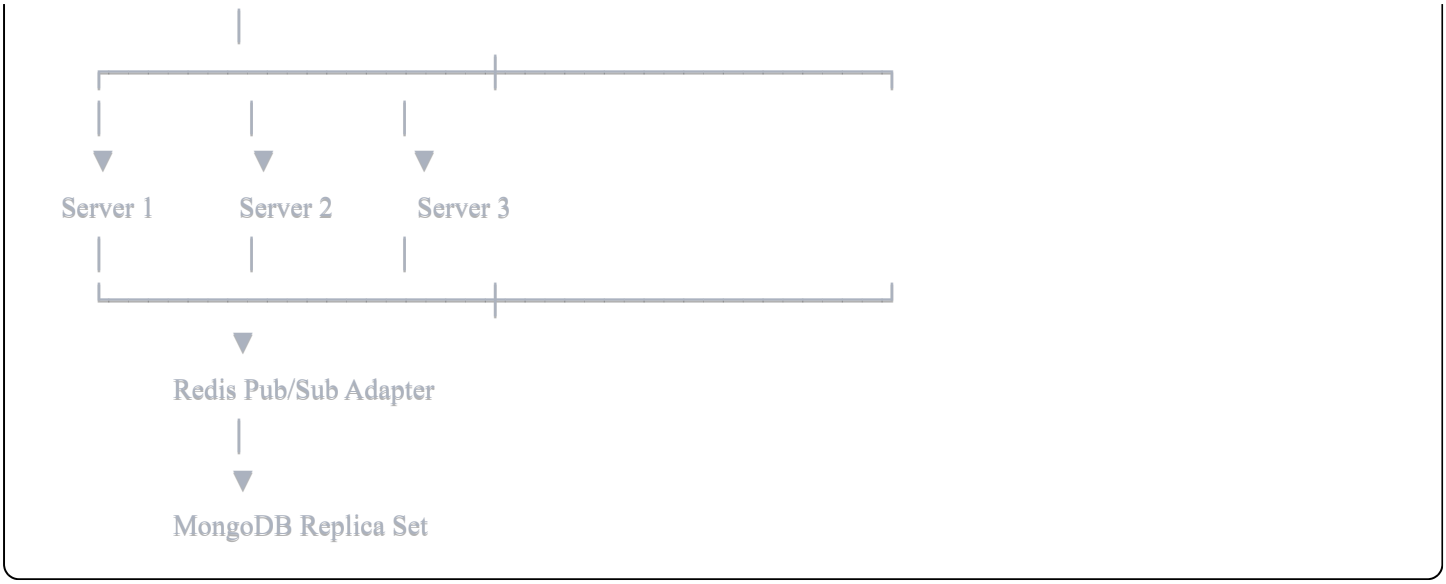
Data relationships use a hybrid approach: embedded last message in conversations provides quick preview; referenced users avoid duplication; denormalized sender information in messages improves display performance.

---

3. Scalability & Performance Architecture

Horizontal Scaling Design





Redis Adapter enables Socket.io across multiple servers through publish-subscribe mechanism. When a user on Server 1 sends a message, Redis broadcasts to all servers, allowing Server 2 to deliver to its connected clients.

MongoDB Replication provides read scaling with primary node handling writes and replica nodes serving reads. Automatic failover ensures high availability. Sharding available for extreme scale scenarios.

Caching Strategy uses Redis to store frequently accessed data (user status, recent messages), reducing database load by 60-80%. Cache invalidation occurs on data updates with TTL preventing stale data.

Performance Optimizations include message pagination (20-50 messages per load), infinite scroll for history, virtual scrolling for smooth rendering, connection pooling for database efficiency, CDN for static assets, and image compression reducing bandwidth by 70%.

---

4. Security Architecture

Authentication & Authorization

JWT Token System implements two-token strategy: short-lived access tokens (15 minutes) for API requests and long-lived refresh tokens (7 days) for session maintenance. Passwords use bcrypt hashing with 12 salt rounds. Token refresh mechanism prevents frequent re-authentication.

API Security Layers include CORS configuration restricting trusted domains, rate limiting preventing brute force (100 requests/minute), input validation sanitizing user data, and protection against SQL injection and XSS attacks through parameterized queries.

WebSocket Security requires authentication during handshake, validates tokens for every connection, implements rate limiting on messages (10 per second), and checks permissions before broadcasting events.

Data Protection enforces HTTPS encryption for all API communication, WSS (WebSocket Secure) for real-time connections, never exposes passwords in responses, and encrypts sensitive data at rest in database.

---

5. Monitoring & Reliability

System Monitoring

Key metrics tracked: active WebSocket connections for load monitoring, message delivery latency (target under 100ms), database query performance (95th percentile), API response times by endpoint, error rates and types, and server resource utilization (CPU, memory, disk).

Logging strategy captures application logs for HTTP requests/responses, Socket events for real-time debugging, error logs with stack traces, security audit logs for authentication attempts, and performance logs identifying bottlenecks.

Reliability Mechanisms

Fault Tolerance provides graceful degradation when WebSocket unavailable, automatic retry with exponential backoff, message queuing for offline scenarios, and circuit breakers preventing cascading failures.

Data Consistency ensures ACID transactions for critical operations, optimistic locking for concurrent updates, message deduplication preventing duplicates, and eventual consistency for non-critical data.

High Availability targets 99.9% uptime (43 minutes downtime/month) through zero-downtime deployments, health checks with automatic restart, and geographic redundancy for global users. Daily automated backups enable point-in-time recovery with 30-day retention.

Conclusion

This system design delivers a scalable, secure, and performant real-time chat application. The architecture supports horizontal scaling for millions of concurrent users, maintains sub-100ms message delivery, implements multi-layered security protection, and provides 99.9% uptime reliability. The modular design enables independent scaling of components and facilitates future feature additions without major refactoring.