ChatFlow: Technical Documentation

---

1) Functional requirements

User management

- Register user (`username`, `email`, `name`, `password`, optional `avatar`). (POST `/auth/register`)
- Login (username + password). JWT returned. (POST `/auth/login`)
- Basic user profile: `username`, `name`, `avatar`, plus `online` and `lastOnline`. (GET `/users/me`; GET `/users`)

Messaging

- Send text messages (one-to-one and group). (POST `/conversations/:id/messages`)
- View conversation history (messages decrypted server-side). (GET `/conversations/:id/messages`)
- Create conversations (DMs and groups). (POST `/conversations`)
- Real-time push when users are online using Socket.IO (server emits `message:new`, `conversation:new`, `presence:update`, and `typing` events).

Chat management

- List all conversations for a user with last-message preview and basic presence info. (GET `/conversations`)
- Create new conversations (DM or group). (POST `/conversations`)

- Basic message timestamps and a minimal message `status` (`sent` / `delivered`). The `delivered` status can be set via POST `/conversations/:id/delivered`.

Bonus features implemented

- **Message status**: `sent` → `delivered` (stored on message documents).
- **Presence**: user `online` boolean + `lastOnline` persisted in the `users` collection and broadcast via `presence:update`.
- **Typing indicator**: `typing` Socket.IO event broadcast to the conversation room.

---

2) Non-functional requirements RESTful API:
Express + TypeScript endpoints (server/src/routes).

- **Realtime**: Socket.IO (WebSockets) for presence, typing, and pushing new messages/conversations. Path configured at `/ws`.
- **DB persistence**: MongoDB (Mongoose).
- **Security basics**: JWT auth middleware; bcrypt password hashing; express-rate-limit.
- **Error handling**: Basic validation and status codes at endpoints.
- **Developer-friendly**: Seed script to create demo users and sample conversations; unit tests (Jest + mongodb-memory-server).

---

3) Assumptions (used for design & numbers)

- Initial deployment is web-first (React client). Mobile clients can connect the same way.

- Early production target: **10,000 daily active users (DAU)** (this is a reasonable early assumption and matches the prior draft).
- Average messages per user per day: **10**.
- Peak hour traffic: assume **10%** of daily messages occur in peak hour.

(See section 5-c for numeric derivations & constraints.)

---

4) API: CRUD operations (by category)

Base URL used in code: `http://localhost:3000` (server `.env.example` shows `PORT=3000`). All authenticated endpoints expect `Authorization: Bearer <token>`.

User Management

POST /auth/register - register a new user

- Request body:

```
{
  "username": "alice",
  "email": "alice@example.com",
  "name": "Alice",
  "password": "password123",
  "avatar": "https://..."
}
```

- Responses:
  - `201 Created` - `{ token: "<jwt>", user: { id, username, name, email, avatar } }`
  - `400` for missing fields

POST /auth/login - authenticate

- Request: `{ "username": "alice", "password": "password123" }`
- Response:
  - `200` - `{ token: "<jwt>", user: {...} }`
  - `401` - invalid credentials

GET /users/me - get profile for logged-in user

- Auth required. Returns user record excluding password.

GET /users - list users (minimal info)

- Auth required. Returns `[{ _id, username, name, online }]` for all users - used by client for presence.

Chat / Conversation Management

GET /conversations - list the user's conversations

- Auth required. Returns conversations the user is a participant of, each with participants (username, name, avatar, online, lastOnline), `isGroup`, and `lastMessage` (decrypted text, timestamp, status). Useful for left-hand convo list.

POST /conversations - create DM or group

- Request:

```
{
  "participantUsernames": ["bob"],    // for DM include one username;
for group include 2+ usernames
  "name": "Project Crew"              // required for groups
}
```

- Response: `201` with the created conversation (participants populated with minimal profile).

GET /conversations/:id/messages - fetch history (sorted by time)

- Auth required. Response: `[{ _id, senderId, createdAt, status, text }]` (server decrypts stored encrypted text).

POST /conversations/:id/messages - send a message (server saves + emits)

- Request: `{ "text": "hello" }`
- Server actions:
    - Stores message in `messages` collection with `textEncrypted` (AES encryption helper), `status: "sent"`, `conversationId`, `senderId`.
    - Emits `message:new` to conversation Socket.IO room with a decrypted `text` payload so connected clients can render immediately.
- Response: `201 { _id, createdAt }`

POST /conversations/:id/delivered - mark `sent` messages as `delivered` for that conversation

- Auth required. Implementation: `Message.updateMany({ conversationId, status: 'sent' }, { $set: { status: 'delivered' }})`
- Response: `{ ok: true }`

GET /conversations/search/all?q=term - naive server-side decrypt-and-search

- Auth required. Server decrypts up to 500 messages for the user's conversations and filters them client-side (simple demo search; not production-scale).

Socket.IO / realtime events

- **Client → Server**
  - Connection: `io("http://localhost:3000", { path: "/ws", auth: { token } })` - token used to authenticate socket, server verifies JWT.
  - `typing` - payload `{ conversationId, username }` (sent while composing).
  - (optional) `message:send` - client can emit instead of using the REST send endpoint; server will broadcast to the room.
- **Server → Clients**
  - `message:new` - when a message is created (payload includes decrypted `text`, `senderId`, `conversationId`, `createdAt`, `status`, `_id`).
  - `conversation:new` - when a new convo is created; server emits to all participants so they can update lists.
  - `presence:update` - incremental presence updates for a user with `{ userId, online, lastOnline }`.
  - `typing` - relayed typing notifications (server broadcasts to the conversation room, excluding sender).

---

5) Data model & DB choice

High-level models (Mongoose)

users (`User` model)

```
{
  _id,
  username: String, // unique
  email: String,    // unique
  name: String,
  avatar?: String,
  passwordHash: String,
  lastOnline?: Date,
  online: Boolean, // default false
  createdAt, updatedAt
}
```

conversations (`Conversation` model)

```
{
  _id,
  participants: [ObjectId of users],
  isGroup: Boolean,
  name?: String,    // for groups
  createdAt, updatedAt
}
```

messages (`Message` model)

```
{
  _id,
  conversationId: ObjectId,
  senderId: ObjectId,
  textEncrypted: String,   // encrypted string stored
  status: "sent" | "delivered", // default 'sent'
  createdAt
}
```

> Note: The code uses the `Conversation` model for both DMs and groups - there is no separate `groups` collection or `users_chats` mapping collection. Each conversation stores its participants array (flexible and well-suited to MongoDB).

Why MongoDB

- **Flexible document model**: `participants` array in `conversations` fits naturally (variable number - DM or group).
- **High write throughput**: chat systems are write-heavy; MongoDB handles many small writes well with appropriate indexing.

- **Easier scaling / sharding**: message collection can be sharded by `conversationId` to partition hot chats.
- **Developer speed**: Mongoose schemas are compact and easy to iterate in a small demo-focused project.

(If you need full relational constraints, Postgres can work - but it'll need schema join tables for many-to-many `users_chats`. For the current design, MongoDB is a pragmatic fit.)

Security & encryption note

- Messages are encrypted on disk with AES-256-CBC using `server/src/utils/crypto.ts` and the `ENC_KEY` env variable. **This is encryption-at-rest / server-side**, not true E2E. The server decrypts to perform search and to include plaintext in `message:new` payloads.

5-c) Estimation & Constraints (traffic, load, events)

These numbers are *explicitly computed from the assumptions*:

- **Users (DAU)**: 10,000
- **Avg messages per user/day**: 10
  - Daily messages = `10,000 × 10 = 100,000 messages/day`
- **Peak hour traffic** (assume 10% of daily messages in peak hour)
  - `100,000 × 0.10 = 10,000 messages` in peak hour
  - Messages per second during peak hour = `10,000 / 3600 ≈ 2.777... ≈ 2.78 messages/sec` (**~3 msg/s**).
- **Typing events**: If 5% of online users type concurrently and each emits ~1 typing event every 2 seconds:
  - Example with 1,000 concurrent users: `0.05 × 1000 = 50 typers`. Typing events/sec ≈ `50 / 2 = 25 events/sec`.

- **Presence heartbeats** (client-driven approach in other designs): client can send heartbeat every 30–60s; in this app presence is handled by socket open/close and server updates `online`/`lastOnline`. If heartbeats are used, for 1000 concurrent users with 30s heartbeat → ~33 pings/sec.

Storage estimate (rough)

- Average encrypted message payload ~200 bytes (depends on message length & base64). For 100k messages/day → ~20 MB/day. Multiply by retention period to plan storage.

Constraints from code

- The `search` endpoint decrypts up to 500 messages in-memory - not suitable for large-scale search. For production use, index/plaintext search must be handled differently (search indexing with sanitized/plaintext tokens, or client-side E2E search protocols if encrypted).

---

6) High-level design & why WebSockets

Architecture (what's in the code)

- **Client**: React (Vite). UI components `Welcome` (login/register) and `ChatLayout` (convo list, message pane). Uses axios for REST calls and `socket.io-client` for realtime updates.

- **API Server**: Node.js + Express + TypeScript exposing REST endpoints for CRUD operations and JWT-based auth.

- **Realtime**: Socket.IO mounted on the same server at path `/ws`. Used for presence, typing, and broadcasting new messages and new conversations to participants.

- **Database**: MongoDB (Mongoose models in `server/src/models`).

- **Message encryption**: AES helper in `server/src/utils/crypto.ts`.

- **Rate limiting**: `express-rate-limit` is wired in middleware.

- **Seed & tests**: `server/src/seed/seed.ts` and Jest tests in `server/tests`.

Why WebSockets / Socket.IO (as in the code)

- Pull Model
    - The client can periodically send an HTTP request to servers to check if there are any new messages. This can be achieved via something like [Long polling]
- Push model
    - The client opens a long-lived connection with the server, and once new data is available it will be pushed to the client. We will use WebSockets for this.
- The pull model approach is not scalable as it will create unnecessary request overhead on our servers and most of the time the response will be empty, thus wasting our resources. To minimize latency, using the push model with WebSockets is a better choice because then we can push data to the client once it's available without any delay, given the connection is open with the client. Also, WebSockets provide full-duplex communication, unlike Server-Sent Events (SSE), which are only unidirectional.
- Chat requires **bi-directional, low-latency** comms (user typing notifications, presence, immediate message delivery). Socket.IO provides this with fallbacks, rooms (used to scope events per conversation), and a friendly API. The code uses rooms - server `socket.join(conversationId)` - so the server can fan out a message only to conversation participants.

---

7) Tradeoffs (choices made in code)

- **Single MongoDB + per-message encryption** (easy dev & demo) vs full E2E encryption (higher security but makes search/mentions harder).

- **In-memory socket mappings** (current `websocket.ts` keeps a map of userId → socketIds) - simple and fast for small clusters but **not** multi-process/multi-instance safe. For scale you need a Socket.IO adapter & Redis/adapter-based pub/sub.

- **No Redis currently** - the code persists presence to MongoDB and broadcasts presence updates. Redis would reduce DB writes and act as shared session/presence store in multi-instance deployments.

- **Search decrypts messages server-side** - easy but expensive and insecure at scale. Replace with indexed plaintext search (controlled access) or move to E2E design with client-side search strategy when required.

---

8) How to scale this application (Scaling considerations)

Short-term (small growth)

1. Add worker pool and keep a single MongoDB replica set. Add sharding for `messages` when message counts grow very large.
2. Use a shared socket adapter: run multiple Node instances behind a load balancer and use `socket.io-redis` adapter (or `socket.io` adapter using Redis/replicated pubsub) so that broadcasts (`message:new`, `conversation:new`, presence) are delivered across instances.
3. Add Redis for presence & session store (store `userId -> socketIds` in Redis). This allows instance failover without losing presence info.

Medium-term

- Use a message broker (Kafka / NATS) for event fan-out between microservices (message ingestion → delivery → push notifications).
- Move media to object storage (S3) + CDN. Add asynchronous background jobs for media processing.
- Add Horizontal DB sharding for `messages` by `conversationId`. Use read replicas for reads (convo lists, history).
- Replace the naive search with an indexing pipeline (Elasticsearch / OpenSearch) that receives plaintext tokens from a controlled worker (if not E2E).

Production concerns

- Sticky sessions or route WebSockets with a load balancer that supports WebSockets (ELB/ALB/Nginx) OR run socket.io with adapter described above.
- Add rate limits per endpoint, request validation, schema validation library.
- Implement monitoring/alerting (Prometheus/Grafana, Sentry).

---

9) How to run locally (quick start)

1. Copy `.env.example` → `server/.env` and set values:

```
PORT=3000
MONGO_URI=mongodb://localhost:27017/chatflow
JWT_SECRET=supersecret
ENC_KEY=0123456789abcdef0123456789abcdef
RATE_LIMIT_WINDOW_MS=60000
RATE_LIMIT_MAX=120
```

2. From the project root:

```
npm install        # installs workspace deps
npm run dev        # runs both server and client concurrently
# or
cd server && npm install && npm run dev
cd client && npm install && npm run dev
```

3. Optionally seed demo data:

```
npm run seed   # runs server/seed/seed.ts to create sample
users/convos/messages
```

4. Tests:

```
npm test   # runs server tests (Jest + mongodb-memory-server)
```

Client

- Connects to server at `http://localhost:3000` by default. The React client uses `io("http://localhost:3000", { path: "/ws", auth: { token } })`.
