Realtime Chat App - Complete Project Handbook

Prepared on: 2026-02-24

1. Executive Summary

This project is a full-stack real-time chat platform built with a clean separation of concerns:

- Frontend: React + Vite + Zustand + Socket.IO client
- Backend: Node.js + Express + PostgreSQL + Socket.IO server
- Auth: JWT session model with DB-backed session validation
- Realtime: room-based messaging, typing, seen status, and notification events
- Additional auth features: Google Sign-In and email verification workflow

The codebase follows a modular, feature-first structure on frontend and controller-route-service layering on backend.

2. Repository Structure

```text
Chat_App/
  backend/
    src/
      config/
      controllers/
      middlewares/
      routes/
      services/
      utils/
    Chat_App.sql
  frontend/
    src/
      app/
      components/
      features/
      providers/
      routes/
      services/
      store/
      styles/
  docs/
  shared/
```

3. Technology Stack and Why It Was Chosen

Backend

- Express: lightweight API layer
- PostgreSQL (‘pg‘): relational consistency for users, rooms, members, messages, statuses
- JWT (‘jsonwebtoken‘): signed stateless tokens
- Socket.IO: realtime bidirectional events with room partitioning

- bcryptjs: password hashing
- Resend API: transactional verification email delivery

Frontend

- React: component-driven UI
- React Router: route guards and page-level auth flow
- Zustand: simple global state (auth/chat) without heavy boilerplate
- Axios: centralized HTTP client and token injection
- Socket.IO client: realtime sync with backend events

4. Backend Module-by-Module Explanation

4.1 Config Layer

‘backend/src/config/env.js‘
Responsibility:
- Reads environment variables
- Applies defaults
- Normalizes multi-origin CORS config
- Exposes auth/email/google config

Key variables:
- ‘CLIENT_ORIGIN‘
- ‘APP_BASE_URL‘
- ‘JWT_SECRET‘, ‘JWT_EXPIRES_IN‘
- ‘GOOGLE_CLIENT_ID‘
- ‘EMAIL_VERIFICATION_TTL_MINUTES‘
- ‘RESEND_API_KEY‘, ‘EMAIL_FROM‘

‘backend/src/config/db.js‘
Responsibility:
- Creates PostgreSQL pool
- Validates DB password presence
- Exposes ‘connectDB()‘ and ‘query()‘

Method used:
- Shared pool for efficiency
- SSL toggle via env for production compatibility

4.2 Middleware Layer

‘backend/src/middlewares/authMiddleware.js‘
Responsibility:
- Extract bearer token
- Verify JWT signature
- Validate active session in ‘user_sessions‘
- Attach ‘req.user‘, ‘req.token‘, ‘req.jwt‘

Method used:
- Two-step auth: cryptographic JWT verification + DB session validity

‘backend/src/middlewares/errorHandler.js‘
Responsibility:
- Unified not-found and error responses
- Logs server errors (‘>=500‘)

4.3 Utility Layer

‘backend/src/utils/jwt.js‘
Methods:
- ‘signToken(payload)‘
- ‘verifyToken(token)‘
- ‘tokenExpiresAt(token)‘

‘backend/src/utils/asyncHandler.js‘
Method:
- Wraps async controllers to forward promise rejections to central error handler

4.4 Routes Layer

‘backend/src/routes/v1/authRoutes.js‘
- ‘POST /auth/signup‘
- ‘POST /auth/login‘
- ‘POST /auth/google‘
- ‘POST /auth/verify-email‘
- ‘POST /auth/resend-verification‘
- ‘GET /auth/me‘
- ‘POST /auth/logout‘

‘backend/src/routes/v1/roomRoutes.js‘
Room listing, private requests, group creation, group joins, room history hide.

‘backend/src/routes/v1/messageRoutes.js‘
Message list/send/edit/delete/seen and room clear operations.

‘backend/src/routes/v1/userRoutes.js‘
User search, connected users list, one-time gender onboarding.

4.5 Controller Layer

Auth Controller (‘backend/src/controllers/auth/authController.js‘)
Main methods:
- ‘signup‘
- ‘login‘
- ‘googleAuth‘
- ‘verifyEmail‘
- ‘resendVerification‘
- ‘me‘
- ‘logout‘

Implementation methods used:
- Password hashing via bcrypt
- Email normalization and username normalization

- Session creation in ‘user_sessions‘
- Google token verification with issuer/audience/expiry checks
- Email verification token hashing (SHA-256)
- Verification URL generation with ‘APP_BASE_URL‘

Room Controller (‘backend/src/controllers/chat/roomController.js‘)
Main methods:
- ‘listRooms‘
- ‘listPublicGroups‘
- ‘createPrivateRoom‘
- ‘respondPrivateRequest‘
- ‘createGroupRoom‘
- ‘requestJoinGroup‘
- ‘deleteGroupRoom‘
- ‘hideRoomHistory‘

Implementation methods used:
- Query-time member aggregation with JSON in SQL
- Private room membership state machine (‘PENDING‘, ‘APPROVED‘, ‘REJECTED‘)
- Event-driven notifications via socket user rooms (‘user:<id>‘)
- Soft behavior for history hide via ‘hidden_at‘

Message Controller (‘backend/src/controllers/message/messageController.js‘)
Main methods:
- ‘listMessagesByRoom‘
- ‘sendMessage‘
- ‘editMessage‘
- ‘deleteMessage‘
- ‘markMessageSeen‘
- ‘clearMessagesByRoom‘

Implementation methods used:
- Membership assertion before each room action
- Per-user message status in ‘message_status‘
- Transactional send flow: insert message + statuses
- Socket emits: ‘message:new‘, ‘message:updated‘, ‘message:deleted‘, ‘message:status‘

User Controller (‘backend/src/controllers/user/userController.js‘)
Main methods:
- ‘searchUsers‘
- ‘listConnectedUsers‘
- ‘setGender‘

Implementation methods used:
- Search via ‘ILIKE‘
- Gender update is one-time lock (‘WHERE gender IS NULL‘)

4.6 Service Layer

Socket Server (‘backend/src/services/socket/socketServer.js‘)
Core behavior:
- Socket auth middleware verifies token + active DB session

- On connect: joins personal room and approved chat rooms
- Handles:
  - ‘typing:update‘
  - ‘room:join‘
  - ‘room:leave‘
  - ‘message:seen‘
  - ‘disconnect‘

Email Service (‘backend/src/services/email/emailService.js‘)
Core behavior:
- Sends verification email via Resend API
- Falls back to console-mode if email config missing

5. Frontend Module-by-Module Explanation

5.1 Bootstrapping and Providers

‘frontend/src/main.jsx‘
- React root + BrowserRouter + AppProviders

‘frontend/src/app/providers/AppProviders.jsx‘
- Composes ‘AuthProvider‘ then ‘SocketProvider‘

‘frontend/src/providers/AuthProvider.jsx‘
- Hydrates auth state from localStorage
- Calls ‘meApi()‘ after hydration to validate server session and refresh profile

‘frontend/src/providers/SocketProvider.jsx‘
- Connects socket when token is present
- Registers socket event listeners
- Syncs events into Zustand chat store

5.2 Routing and Guards

‘frontend/src/routes/index.jsx‘
Public routes:
- ‘/‘ landing page
- ‘/login‘
- ‘/signup‘
- ‘/verify-email‘

Protected routes:
- ‘/onboarding/gender‘
- ‘/chat‘
- ‘/profile‘

Guards:
- ‘PublicOnlyRoute‘
- ‘ProtectedRoute‘
- ‘GenderSetupOnlyRoute‘
- ‘RequireGenderRoute‘

5.3 State Management

‘frontend/src/store/authStore.js‘
State:
- user
- token
- isAuthenticated
- hydrated

Methods:
- hydrate from storage
- login/logout
- update profile

‘frontend/src/store/chatStore.js‘
State:
- chats
- ‘messagesByChat‘
- active chat id
- typing indicators
- notifications

Methods:
- upsert chat/message
- message status updates
- typing updates
- notification queue

5.4 Feature Modules

Auth Feature
Files:
- ‘features/auth/pages/*‘
- ‘features/auth/hooks/useAuthActions.js‘
- ‘features/auth/api/authApi.js‘
- ‘features/auth/components/*‘

Capabilities:
- Login/signup
- Google popup sign-in
- Email verification page + resend
- Conditional redirect when login fails with ‘requiresEmailVerification‘

Chat Feature
Files:
- ‘features/chat/pages/ChatPage.jsx‘
- ‘features/chat/api/chatApi.js‘

Capabilities:
- Load rooms, groups, users
- Create private/group chat
- Join groups

- Hide room history
- Clear chat

Message Feature
Files:
- ‘features/messages/api/messageApi.js‘
- ‘components/messages/*‘

Capabilities:
- Send/edit/delete messages
- Message type tags (TEXT/IMAGE/FILE/VOICE)
- Seen updates
- Ctrl/Cmd+Enter send behavior

Search Feature
Files:
- ‘components/search/UserSearch.jsx‘
- ‘features/search/api/searchApi.js‘

Capabilities:
- Search users
- Search loaded local message text
- Start private room directly from search results

Notifications Feature
Files:
- ‘components/notifications/*‘

Capabilities:
- Toast stack
- Alert panel with grouped message notifications
- Request panel for private chat approvals/rejections

5.5 UI and Layout

- ‘components/layout/MainLayout.jsx‘ handles top navigation and user actions
- ‘components/layout/AuthLayout.jsx‘ handles auth page split layout and mobile promo
- ‘features/auth/pages/LandingPage.jsx‘ provides animated marketing entry page

6. Data Model and Database Design

File: ‘backend/Chat_App.sql‘

Core tables:
- ‘users‘
- ‘rooms‘
- ‘room_members‘
- ‘messages‘
- ‘message_status‘
- ‘user_sessions‘
- ‘media‘

Notable columns:
- ‘users.email_verified‘
- ‘users.email_verification_token_hash‘
- ‘users.email_verification_expires_at‘

Design methods used:
- UUID PKs for distributed-safe identifiers
- Join table for room membership with role and status
- Separate table for per-user message state
- Session table to allow token invalidation/logout

7. Feature-to-Implementation Mapping

7.1 JWT + Session Authentication
Methods used:
- JWT signing (‘signToken‘)
- DB session persistence (‘user_sessions‘)
- Middleware validation by token + session row

Outcome:
- Tokens can be revoked on logout
- Socket auth and API auth share same session validity rules

7.2 Google Sign-In
Methods used:
- Browser receives credential via Google JS SDK
- Backend validates with Google tokeninfo endpoint
- Checks issuer, audience, expiration, verified email

Outcome:
- New user auto-created or existing user signed in
- Google users marked email-verified by default

7.3 Email Verification Flow
Methods used:
- Secure random token generation
- SHA-256 token hashing before DB store
- Expiry check with TTL minutes
- Verification and resend endpoints
- Resend email provider integration

Outcome:
- Password login is blocked until verified
- Verification links point to frontend verify page

7.4 Private Request-Based Direct Chat
Methods used:
- Room member statuses (‘PENDING‘, ‘APPROVED‘, ‘REJECTED‘)
- Request/response APIs
- Socket notifications to target/requester

Outcome:

- Controlled access for private 1-to-1 chats

7.5 Group Chat Management
Methods used:
- Public group rooms with member rows
- Admin-only moderation operations
- Join flow with membership updates

Outcome:
- Open community chat model with server-side permission checks

7.6 Realtime Messaging and Presence
Methods used:
- Socket rooms for user and chat channels
- Typing broadcast events
- Seen acknowledgements and status updates
- Online/offline presence updates on connect/disconnect

Outcome:
- Live chat UX with low-latency updates

7.7 Notification and Unread Strategy
Methods used:
- Server pushes ‘notification:new‘
- Client groups notifications by room/sender
- Unread count derived from in-memory notifications

Outcome:
- WhatsApp-like unread cues in sidebar and alert panel

7.8 Responsive UX and Marketing Entry
Methods used:
- CSS media queries for auth/chat pages
- Mobile-aware ordering of chat sections
- Dedicated animated landing page at ‘/‘

Outcome:
- Better mobile readability and a stronger first impression

8. Important Code Snippets (High-Impact Features)

Snippet 1: Email verification token creation (backend)
File: ‘backend/src/controllers/auth/authController.js‘

‘‘‘js
function createEmailVerificationToken() {
  const token = crypto.randomBytes(32).toString("hex");
  const tokenHash = crypto.createHash("sha256").update(token).digest("hex");
  const ttlMinutes = Math.max(MIN_EMAIL_VERIFICATION_MINUTES,
  Number(EMAIL_VERIFICATION_TTL_MINUTES) || 0);
  const expiresAt = new Date(Date.now() + ttlMinutes * 60 * 1000);
  return { token, tokenHash, expiresAt };

}
```

### Snippet 2: Block login until email is verified
File: `backend/src/controllers/auth/authController.js`

```js
if (!user.email_verified) {
  return res.status(403).json({
    message: "Email is not verified. Please verify your email before login.",
    requiresEmailVerification: true,
    email: user.email,
  });
}
```

Snippet 3: Google token verification hard checks
File: `backend/src/controllers/auth/authController.js`

```js
if (!GOOGLE_ISSUERS.has(issuer)) {
  const error = new Error("Invalid Google token issuer");
  error.status = 401;
  throw error;
}
if (allowedAudiences.length > 0 && !allowedAudiences.includes(audience)) {
  const error = new Error("Google token audience mismatch");
  error.status = 401;
  throw error;
}
```

Snippet 4: Transactional message send + status fanout
File: `backend/src/controllers/message/messageController.js`

```js
const messageResult = await client.query(
  `INSERT INTO messages (room_id, sender_id, encrypted_content, message_type)
   VALUES ($1, $2, $3, $4)
   RETURNING message_id, room_id, sender_id, encrypted_content, message_type, is_edited,
   is_deleted, created_at`,
  [roomId, req.user.user_id, text.trim(), type]
);
// then insert message_status for all approved room members
```

Snippet 5: Socket auth middleware uses DB session check
File: `backend/src/services/socket/socketServer.js`

```js
io.use(async (socket, next) => {
  const token = socket.handshake.auth?.token;
```
```

```
  const decoded = verifyToken(token);
  const userResult = await query(
    'SELECT u.user_id FROM users u
     JOIN user_sessions s ON s.user_id = u.user_id
     WHERE u.user_id = $1 AND s.jwt_token = $2 AND s.is_valid = TRUE AND s.expires_at > NOW()'
  );
  if (userResult.rowCount === 0) return next(new Error("Unauthorized"));
  next();
});
```

Snippet 6: Frontend handles unverified-login redirect
File: `frontend/src/features/auth/hooks/useAuthActions.js`

```js
const requiresEmailVerification = Boolean(err?.response?.data?.requiresEmailVerification);
if (requiresEmailVerification) {
  const email = String(err?.response?.data?.email || payload?.email || "").trim();
  navigate(`/verify-email?email=${encodeURIComponent(email)}`);
  return;
}
```

Snippet 7: Chat store message upsert for realtime sync
File: `frontend/src/store/chatStore.js`

```js
upsertMessage: (message) => {
  set((state) => {
    const existing = state.messagesByChat[message.chatId] || [];
    return {
      messagesByChat: {
        ...state.messagesByChat,
        [message.chatId]: upsertById(existing, message),
      },
    };
  });
},
```

Snippet 8: Composer Enter key behavior fix
File: `frontend/src/components/messages/MessageComposer.jsx`

```js
if (event.key === "Enter" && (event.ctrlKey || event.metaKey)) {
  event.preventDefault();
  handleSend();
}
```

9. API Contract Summary

Auth:
- ‘POST /api/v1/auth/signup‘
- ‘POST /api/v1/auth/login‘
- ‘POST /api/v1/auth/google‘
- ‘POST /api/v1/auth/verify-email‘
- ‘POST /api/v1/auth/resend-verification‘
- ‘GET /api/v1/auth/me‘
- ‘POST /api/v1/auth/logout‘

Users:
- ‘GET /api/v1/users/search‘
- ‘GET /api/v1/users/connected‘
- ‘PATCH /api/v1/users/gender‘

Rooms:
- ‘GET /api/v1/rooms‘
- ‘GET /api/v1/rooms/groups/public‘
- ‘POST /api/v1/rooms/private‘
- ‘GET /api/v1/rooms/private/requests‘
- ‘PATCH /api/v1/rooms/private/requests/:roomId‘
- ‘POST /api/v1/rooms/group‘
- ‘PATCH /api/v1/rooms/:roomId/hide‘
- ‘POST /api/v1/rooms/:roomId/join-request‘
- ‘GET /api/v1/rooms/group/requests‘
- ‘PATCH /api/v1/rooms/group/:roomId/requests/:userId‘
- ‘DELETE /api/v1/rooms/group/:roomId‘

Messages:
- ‘GET /api/v1/messages/room/:roomId‘
- ‘PATCH /api/v1/messages/room/:roomId/clear‘
- ‘POST /api/v1/messages‘
- ‘PATCH /api/v1/messages/:messageId‘
- ‘DELETE /api/v1/messages/:messageId‘
- ‘PATCH /api/v1/messages/:messageId/seen‘

10. Security Notes and Current Gaps

Implemented
- Password hashing
- JWT signature checks
- DB-backed session revocation
- Google token issuer/audience validation
- Email token hashing before storage
- Membership checks before room/message actions

Improvement opportunities
- Message payload is stored in ‘encrypted_content‘ column but currently uses plain text from API input
- No explicit rate limiting middleware yet
- No centralized request validation library (e.g., Zod/Joi) yet
- ‘.env‘ loading currently uses ‘override: true‘ and can override platform vars in some deployments

11. Deployment Notes

Frontend env:
- ‘VITE_API_BASE_URL‘
- ‘VITE_SOCKET_URL‘
- ‘VITE_GOOGLE_CLIENT_ID‘

Backend env:
- DB variables
- ‘CLIENT_ORIGIN‘
- ‘JWT_SECRET‘
- ‘GOOGLE_CLIENT_ID‘
- ‘APP_BASE_URL‘
- ‘EMAIL_VERIFICATION_TTL_MINUTES‘
- ‘RESEND_API_KEY‘
- ‘EMAIL_FROM‘

Recommended production values:
- ‘CLIENT_ORIGIN‘ include all frontend origins (comma-separated)
- ‘APP_BASE_URL‘ should point to frontend domain with verify page

12. Common Interview Questions with Model Answers

Q1. Why use both JWT verification and session table checks?
A: JWT validates signature and expiry, but session-table checks allow server-side revocation and logout invalidation. This combines stateless auth speed with operational control.

Q2. Why is ‘message_status‘ separated from ‘messages‘?
A: Read/seen state is per user, not per message globally. A join table supports independent status for each participant.

Q3. How do you prevent unauthorized socket connections?
A: Socket middleware verifies JWT and checks active, unexpired session row in ‘user_sessions‘ before attaching ‘socket.user‘.

Q4. Why hash email verification tokens before storing?
A: If DB leaks, raw tokens cannot be reused. Verification compares hash(token) to stored hash.

Q5. How is private chat approval modeled?
A: Through ‘room_members.status‘ transitions (‘PENDING‘ to ‘APPROVED/REJECTED‘) in PRIVATE rooms.

Q6. What ensures only room members can read/send messages?
A: ‘assertMembership(roomId, userId)‘ gate in message controller methods.

Q7. Why use Socket.IO rooms?
A: Efficient scoped broadcast to specific chat participants (‘roomId‘) and personal channels (‘user:<id>‘).

Q8. How do you handle unread counts on client?
A: Notifications are stored in Zustand; counts are derived by grouping message notifications by room id.

Q9. How do you keep auth state after refresh?
A: ‘authStore‘ hydrates from localStorage via ‘authStorage‘, then AuthProvider revalidates with ‘meApi‘.

Q10. Why use Zustand instead of Redux here?
A: Smaller API surface, less boilerplate, and sufficient for medium-complexity chat state.

Q11. What are the tradeoffs of storing notifications only in memory?
A: Fast and simple but not persistent across full refresh or multi-device sessions.

Q12. How do you avoid duplicate usernames for Google users?
A: ‘ensureUniqueUsername‘ loops with random suffix until an unused value is found.

Q13. How are CORS and Socket origins controlled?
A: Backend parses ‘CLIENT_ORIGIN‘ and uses same origin list in Express CORS and Socket.IO CORS config.

Q14. How does the app support group membership updates in realtime?
A: Server emits ‘room:available‘ to specific users after approval or join.

Q15. What would you scale first for high traffic?
A: Add Redis adapter for Socket.IO, add DB indexes for high-frequency queries, and introduce caching for room lists.

Q16. How is one-time gender setup enforced?
A: SQL update includes ‘WHERE gender IS NULL‘, preventing further changes.

Q17. Why keep ‘hidden_at‘ and ‘cleared_at‘ in ‘room_members‘?
A: These are user-room scoped behaviors, not room-global properties.

Q18. How does the app avoid stale socket listeners?
A: ‘SocketProvider‘ registers handlers in effect and cleans them up in return function.

Q19. What are failure modes in verification email flow?
A: Missing provider config, API provider errors, expired tokens, invalid tokens, or mismatched ‘APP_BASE_URL‘.

Q20. How would you harden this project for enterprise use?
A: Add schema migrations, typed validation, rate limits, audit logs, encryption-at-rest strategy, and integration tests.

13. Suggested Next Enhancements

1. Implement true end-to-end encryption or server-side encryption wrapper for message payloads.
2. Add migration tooling (e.g., Knex/Prisma migrations).
3. Add unit/integration tests for auth and message workflows.
4. Add rate limiting and login attempt throttling.
5. Add persisted notification center and read receipts pagination.

End of handbook.