Real-Time Admin Dashboard Project Documentation

A Full-Stack Backend Project Using Node.js, Socket.IO, TypeScript, MongoDB, and Express

---

Table of Contents

---

1. Introduction & Project Overview

This project builds a robust, event-driven real-time backend system designed specifically for modern administrative dashboards and interactive communication workflows. It leverages a combination of powerful technologies:

- **Node.js** and **Express** to build a scalable, performant RESTful API layer.
- **Socket.IO** to implement real-time bidirectional communication for live user interactions.
- **MongoDB** to store structured and unstructured data with high flexibility and performance.
- **TypeScript** to enhance development productivity with static typing, IDE support, and early bug detection.

The system is highly modular and designed with future extensibility in mind, making it ideal for startups, enterprise dashboards, or scalable SaaS platforms.

Key Use Cases and Functionalities

- **Order Tracking and Real-Time Status Updates**: The admin can monitor new orders in real-time and update their status, which reflects immediately on the user's dashboard.
- **One-to-One Messaging System**: Users and admins can engage in secure private conversations with full chat history support.
- **Typing Indicators**: While messaging, users can see when the other party is actively typing, improving the chat experience.
- **Product Broadcast Notifications**: When a new product is added or updated, all connected clients receive instant notifications.
- **Online/Offline Detection**: The system tracks user presence, allowing admins to view which users are online, recently active, or offline.

- **Modular API Layer**: All operations like registration, login, order management, and chat are accessible through a versioned, secure RESTful API.
- **Token-Based Auth with WebSocket Compatibility**: Secure authentication ensures consistent user sessions across HTTP and WebSocket protocols.

---

2. Project Objectives

The primary goal of this project is to create a reliable, scalable, and secure real-time backend system. Below is a comprehensive breakdown of the core objectives and their purposes:

Objective	Description
Real-time backend	Enable low-latency two-way communication using WebSockets via Socket.IO.
Strong typing	Utilize TypeScript to reduce runtime errors and improve code readability.
Flexible data model	Use MongoDB and Mongoose to handle dynamic, evolving data structures.
RESTful operations	Build standardized CRUD APIs for user, product, order, and message entities.
Real-time updates	Implement features like live chat, typing indicators, order notifications.
Secure authentication	Use JWT for stateless, secure API and WebSocket authentication.
Scalable architecture	Design for scalability with microservices compatibility and cloud readiness.
Maintainability	Follow clean code principles, modular structure, and consistent conventions.
Developer experience	Provide robust tooling, dev scripts, and logs for better development workflow.

These objectives ensure the system is well-suited for production environments, supports future feature expansion, and delivers a seamless experience to both users and administrators.

3. Target Audience

This project is ideal for a broad range of developers and product teams who need to build real-time applications with dynamic communication, fast feedback loops, and backend reliability:

- **Backend Engineers** exploring real-time architectures, event-driven systems, and WebSocket-based data streaming.
- **Startup Developers** needing agile, lightweight admin dashboards that monitor user actions, track orders, and enable real-time control without third-party tools.
- **Freelance Developers** building scalable platforms for clients in industries like eCommerce, logistics, education, or live support, where real-time features are essential.
- **Cross-functional Product Teams** integrating live chat, notifications, presence indicators, and push-based updates into modern web or mobile apps.
- **Technical Interns and CS Students** who want to showcase a complete backend solution using modern technologies like TypeScript, Socket.IO, and MongoDB in their portfolios.
- **Platform Architects and CTOs** seeking modular backend architectures that support both REST and WebSocket protocols for seamless client integrations.
- **Companies Building SaaS Tools** that require user activity visibility, administrator intervention, or live analytics for better operational efficiency.

- **DevOps and SREs** aiming to deploy and scale real-time services across distributed infrastructures using containerization and cloud services.

4. Technology Stack

Backend

- Node.js
- Express.js
- Socket.IO
- MongoDB + Mongoose
- TypeScript
- JWT, Bcrypt.js
- Winston, Morgan

Dev Tools

- Postman
- MongoDB Compass
- VSCode
- Git + GitHub

---

5. Project Planning

This section outlines a step-by-step development plan broken into phases, ensuring that each critical component of the real-time backend system is built, tested, and deployed systematically.

Phase 1: Project Setup

- Initialize the Node.js backend with TypeScript, Express, and MongoDB.
- Configure project structure using `src/` folder separation (controllers, routes, models, services).
- Set up development tooling:
  - **ESLint** for linting
  - **Prettier** for code formatting
  - **Husky** + **Lint-staged** for pre-commit hooks
- Setup `.env` and environment-based config management.
- Version control initialized with Git.

Phase 2: Authentication Module

- Implement `/auth/register` and `/auth/login` endpoints.
- Use **bcrypt.js** for password hashing.
- Issue **JWT tokens** upon successful login.
- Add middleware for JWT verification (`auth.middleware.ts`).
- Implement role-based access control for admin/user routes.
- Protect Socket.IO connections with token-based authentication.

Phase 3: Admin Dashboard & Realtime Events

- Integrate **Socket.IO** into the Express server.
- Track and store online users using `socket.id` + `userId`.
- Enable event rooms for admin, individual users.
- Implement:
  - `user_online`, `user_offline` events

- `new_order`, `order_status_update` broadcasts
- `new_product` push notifications to all connected clients

Phase 4: Real-Time Chat System

- Create **Message** model in MongoDB (sender, receiver, text, seen, createdAt).
- Implement `send_message`, `receive_message` events.
- Add support for:
  - **Typing indicators** (`start_typing`, `stop_typing`)
  - **Message seen status**
  - **Chat history API** to fetch past messages
- Handle edge cases: offline users, reconnections, delivery queue.

Phase 5: Testing & Debugging

- Unit test APIs using **Jest**.
- Integration tests using **Supertest** (e.g., auth, CRUD endpoints).
- Socket testing with mocked clients.
- Centralized error handling middleware.
- Enable advanced **logging** with **Winston** and HTTP logging via **Morgan**.
- Add runtime monitoring using **PM2**, **Grafana**, or **Sentry** (optional).

Phase 6: Deployment

- Dockerize the application with multi-stage builds.
- Use **Docker Compose** for local MongoDB and app setup.

- Deploy on cloud providers:
    - **Render**, **Railway**, or **Fly.io** for simplicity
    - **AWS EC2** or **VPS** for full control
- Use **MongoDB Atlas** for production database.
- Enable environment-based build config and secrets injection.
- Set up CI/CD pipeline for automated deployment (optional).

- Seen, typing indicators

Phase 5: Testing & Debugging

- Jest, Supertest
- Logging + monitoring

Phase 6: Deployment

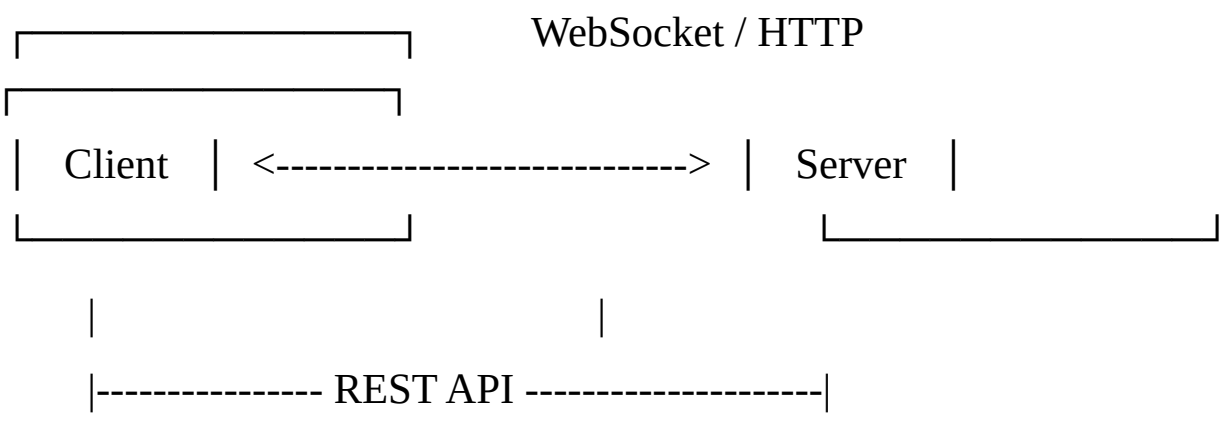
- Docker, Render/Railway

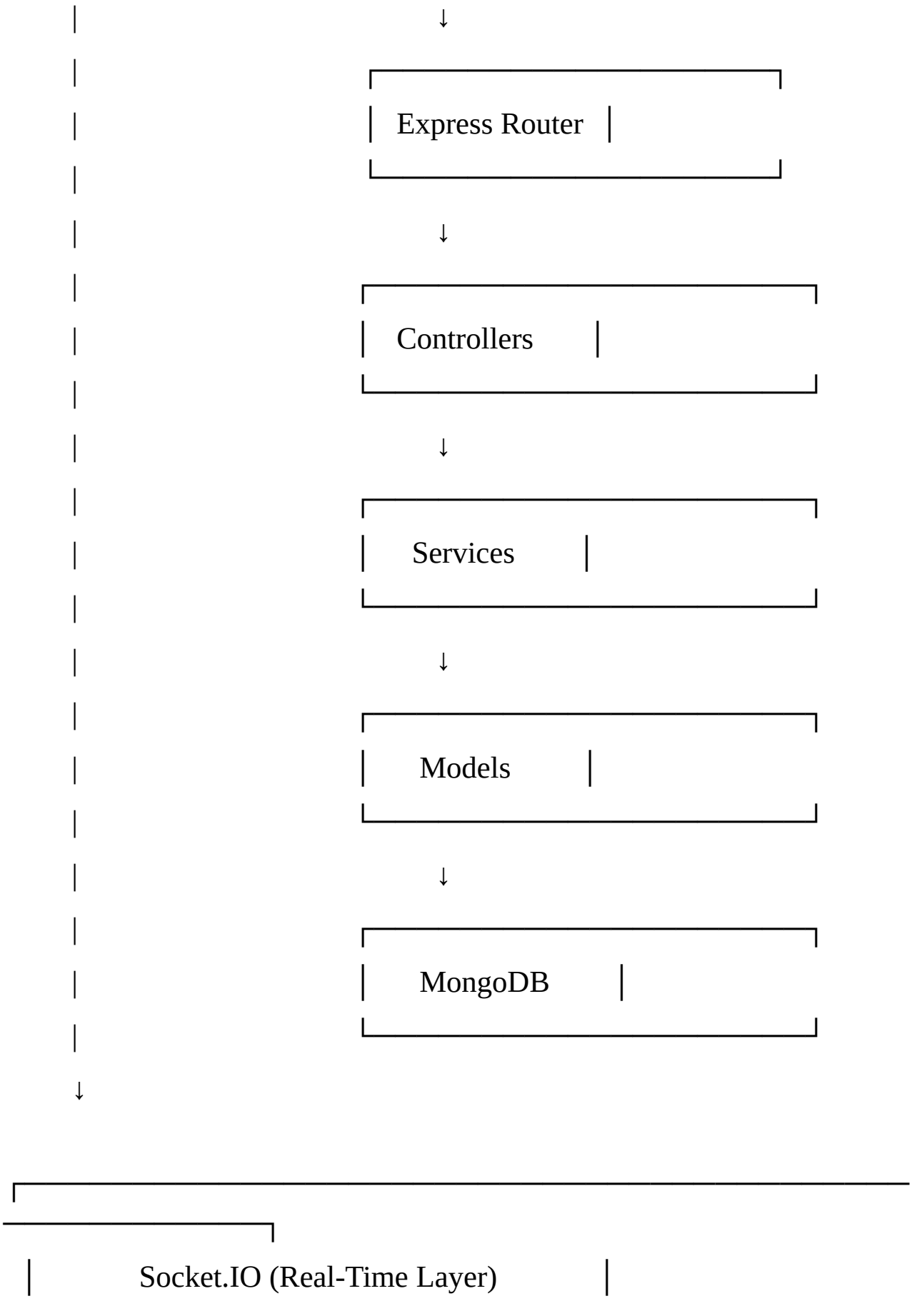
---

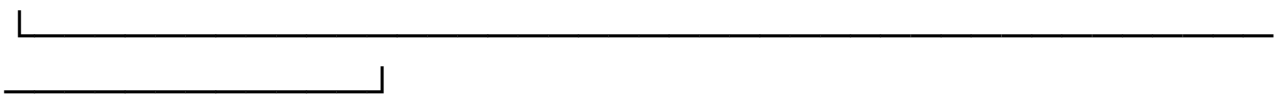
6. System Architecture

The system follows a hybrid **REST + WebSocket** architecture, enabling both stateless HTTP-based communication and persistent real-time connections for live features.

High-Level Overview




Express Router
Controllers
Services
Models
MongoDB
Socket.IO (Real-Time Layer)



Emits/Receives Handles client events

Component Breakdown

- **Client**: Can be a React, Flutter, or any JS-based front end that connects via REST and WebSocket to the backend.
- **Socket.IO**: Manages persistent real-time connections. Listens for events like `send_message`, `order_status_update`, and broadcasts changes.
- **Express REST API**: Used for traditional HTTP requests such as login, registration, fetching products, submitting orders, and retrieving messages.
- **Controllers**: Act as bridges between routes and business logic. They validate requests, call services, and return responses.
- **Services (Optional Layer)**: Encapsulate business logic like querying MongoDB, hashing passwords, or verifying JWTs.
- **Models**: Mongoose schemas that define how data is structured in MongoDB (e.g., User, Message, Order, Product).
- **MongoDB**: NoSQL database used to store users, orders, messages, and system logs in flexible collections.

Real-Time Communication Flow

1. A client connects to the backend via Socket.IO and authenticates using JWT.
2. Upon events like order placement, message sending, or user typing, the server broadcasts events to relevant recipients.

3. The server also handles disconnects and presence detection using socket lifecycle events.
4. At the same time, RESTful endpoints serve CRUD operations and data retrieval.

---

7. Database Design (MongoDB)

The application uses **MongoDB** as a NoSQL document database, managed via **Mongoose** (an Object Data Modeling library for MongoDB and Node.js). It offers flexible schema design, making it ideal for real-time applications where data shapes may evolve.

Collections & Schemas

`users` Collection

Stores registered users (both admins and regular users).

```
{
  _id: ObjectId,
  name: string,
  email: string,
  password: string,
  role: "user" | "admin",
  socketId?: string,        // For tracking real-time connections
  lastActive: Date          // Timestamp of last socket activity
}
```

`Indexes`:

- `email`: unique, for login
- `socketId`: optional index for quick lookup of online users

`orders` Collection

Tracks all orders placed by users and updated by admins.

```
{
  _id: ObjectId,
  userId: ObjectId,          // Reference to User
  productId: ObjectId,       // Reference to
Product
  status: "pending" | "processing" | "shipped" |
"delivered",
  createdAt: Date,
  updatedAt: Date
}
```

`Indexes`:

- `userId`: to retrieve orders by user
- `status`: for filtering by order progress

`products` Collection

Stores the product inventory visible to users/admins.

```
{
  _id: ObjectId,
  name: string,
  price: number,
  stock: number,
  description: string,
  createdAt: Date,
  updatedAt: Date
}
```

`Indexes`:

- `name`: for search queries
- `price`: for filtering and sorting

`messages` Collection

Stores real-time chat messages between users and admins.

```
{
  _id: ObjectId,
  from: ObjectId,            // Sender ID
  to: ObjectId,              // Recipient ID
  text: string,
  seen: boolean,
  createdAt: Date
}
```

`Indexes`:

- Compound index on `{ from, to }` for message threads
- `createdAt`: for chronological sorting

Relationships & References

- `userId`, `from`, and `to` fields are stored as `ObjectId` references.
- Mongoose's `populate()` can be used to join user details when fetching orders or messages.

Validation & Defaults

- Use **Mongoose schema validation** (e.g., required fields, enum constraints for `status`).
- Add timestamps using `timestamps: true` option in schema definitions.
- Apply **pre-save hooks** for hashing passwords and updating `lastActive`.

---

8. REST API Design

The backend offers a secure, RESTful interface for managing authentication, products, orders, and chat history. JWT authentication protects sensitive endpoints, ensuring only authorized users (especially admins) can access or mutate data.

Authentication Endpoints

Endpoint	Method	Auth Required	Description
`/auth/register`	POST	No	Register a new user account
`/auth/login`	POST	No	Authenticate user and return JWT

Product Endpoints

Endpoint	Method	Auth Required	Role	Description
`/products`	GET	No	Any	Get full product list
`/products`	POST	Yes	Admin only	Add new product to catalog
`/products/:id`	PUT	Yes	Admin only	Update product details
`/products/:id`	DELETE	Yes	Admin only	Remove a product

Order Endpoints

Endpoint	Method	Auth Required	Role	Description
`/orders`	GET	Yes	Admin only	Fetch all orders
`/orders/my`	GET	Yes	User only	Retrieve logged-in user’s orders
`/orders`	POST	Yes	User only	Place a new order
`/orders/:id`	PUT	Yes	Admin only	Update order status (e.g., shipped, etc)

Chat Endpoints

Endpoint	Method	Auth Required	Description
`/chat/messages`	GET	Yes	Fetch all messages for logged-in user
`/chat/messages/:id`	GET	Yes	Fetch all messages between user and recipient `id`

Notes

- All protected routes require the JWT to be sent in the `Authorization: Bearer <token>` header.
- Input validation is enforced using `express-validator` or a schema tool like Zod (optional).
- Proper status codes (`200`, `201`, `400`, `401`, `403`, `404`) are used consistently.

---

9. WebSocket Implementation (Socket.IO)

This project integrates **Socket.IO** to enable real-time, bidirectional communication between the backend and clients. Socket.IO runs alongside the Express server and shares the same authentication strategy (JWT) to ensure secure, session-aware communication.

Event-Driven Communication Model

The server listens for and emits events over persistent WebSocket connections. All connected clients are authenticated and managed in a user-to-socket mapping, which allows the system to:

- Emit targeted messages to a specific user or admin
- Broadcast updates to all clients or specific rooms
- Track online/offline status in real time

WebSocket Events Table

Event Name	Direction	Description
`connect`	Client → Server	When a client connects to the server
`disconnect`	Client → Server	Fired when client disconnects
`new_order`	Server → Admin	Notifies admin of a newly placed order
`order_status_update`	Server ↔ Client	Updates users when an order status changes
`send_message`	Client → Server	Client sends a chat message
`receive_message`	Server → Client	Server delivers a message to the recipient
`typing`	Client → Server	Client indicates the user is typing
`user_typing`	Server → Client	Sent to recipient while the other is typing
`user_online`	Server → Client	Announces when a user comes online
`user_offline`	Server → Client	Announces when a user goes offline
`new_product`	Server → Client	Broadcasts that a new product was added

Example: Socket.IO Integration on Server

```
import { Server } from "socket.io";
import jwt from "jsonwebtoken";

const io = new Server(httpServer, {
  cors: { origin: "*" }
});

io.use((socket, next) => {
  const token = socket.handshake.auth.token;
  try {
    const user = jwt.verify(token,
process.env.JWT_SECRET!);
    socket.data.user = user;
    next();
  } catch (err) {
    next(new Error("Authentication failed"));
  }
});

io.on("connection", (socket) => {
  const userId = socket.data.user.id;

  // Notify all clients
  io.emit("user_online", { userId });

  socket.on("send_message", async (msg) => {
    // save to DB, then emit
    const saved = await Message.create(msg);
    io.to(msg.to).emit("receive_message", saved);
  });

  socket.on("typing", (receiverId) => {
    io.to(receiverId).emit("user_typing", { from:
userId });
  });
```

```
  socket.on("disconnect", () => {
    io.emit("user_offline", { userId });
  });
});
```

Authentication with JWT over Socket.IO

Clients must connect using:

```
const socket = io("https://your-server", {
  auth: {
    token: "your-jwt-token"
  }
});
```

---

10. Authentication Flow

The backend uses **JWT-based authentication** to secure both HTTP and WebSocket connections. This ensures that only verified users (admins or clients) can access protected APIs and establish real-time communication channels.

Step-by-Step Authentication Lifecycle

1. **User Submits Login Form**
   The client sends a `POST /auth/login` request with `email` and `password`.
2. **Server Verifies Credentials**
   - Finds the user in MongoDB.
   - Verifies the password with `bcrypt.compare()`.
   - If valid, generates a signed JWT token.
3. **JWT Token Issued**

- The token includes payload fields like `id`, `email`, and `role`.
- Example:

```
{
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6...",
  "user": {
    "id": "64a98f9...",
    "role": "admin"
  }
}
```

4. **Client Stores the Token**
   - JWT is saved on the frontend (usually in `localStorage` or `secure cookies`).
5. **Protected HTTP Calls Use JWT**
   - API requests include the token in the `Authorization` header:

     ```
     Authorization: Bearer <token>
     ```
6. **Middleware Validates Token**
   - A middleware checks the `Authorization` header.
   - If valid, attaches the user to `req.user` and continues.
   - If invalid/expired, returns `401 Unauthorized`.
7. **Socket.IO Auth via Handshake**
   - When the client connects to WebSocket:

     ```
     const socket = io("https://server-url", {
       auth: { token: "JWT_TOKEN_HERE" }
     });
     ```
   - The server verifies this token before allowing connection.

Token Handling Tips

- Use short-lived JWTs (e.g., 15 minutes) with optional refresh token logic.
- Never store JWTs in `localStorage` for highly sensitive apps (consider `httpOnly` cookies).
- Reconnect sockets using the same token when refreshing pages.

Sequence Diagram

```
[Client] ── Login Form ──> [Server]
        <── JWT Token ───

[Client] ── Auth Header ──> [Express Middleware]
        <── Protected Data ──

[Client] ── Socket.IO Connect (with JWT) ──>
[Socket.IO Middleware]
        <── Connection Established ──
```

---

11. Core Features Implementation

This section outlines the key real-time features implemented with **Socket.IO** and MongoDB integration, accompanied by concise example code snippets.

Real-Time Order Updates

- **Functionality**: When a user places a new order, the admin receives an instant notification. When the admin updates the order status, the user is notified in real time.

```
// Notify admin of new order
io.to(adminSocketId).emit("new_order", order);

// Listen for order status update and notify user
socket.on("update_order_status", (order) => {
  io.to(order.userId).emit("order_status_update",
order);
});
```

Chat System

- **Functionality**: Enables one-to-one messaging. Messages are stored in MongoDB and sent immediately to the recipient.

```
socket.on("send_message", async (msg) => {
  const saved = await Message.create(msg);
  io.to(saved.to).emit("receive_message", saved);
});
```

Typing Indicator

- **Functionality**: Shows when a user is typing a message, improving communication responsiveness.

```
socket.on("start_typing", (recipientId) => {
  io.to(recipientId).emit("user_typing", { from:
socket.user.id });
});
```

Product Broadcast

- **Functionality**: When an admin adds a new product, all connected clients receive an instant update.

```
io.emit("new_product", newProduct);
```

Online Presence Detection

- **Functionality**: Tracks user connections and disconnections in real time, broadcasting online/offline status to all clients.

```
socket.on("connect", () => {
  io.emit("user_online", { userId:
socket.user.id });
});

socket.on("disconnect", () => {
  io.emit("user_offline", { userId: socket.user.id
});
});
```

---

12. TypeScript Models & Interfaces

To ensure strong typing and schema consistency, we use **TypeScript interfaces** alongside **Mongoose schemas**.

Interface Example

```
import { Types } from "mongoose";

interface IMessage {
  from: Types.ObjectId;
  to: Types.ObjectId;
  text: string;
  seen: boolean;
  createdAt?: Date;
```

```
}
```

Mongoose Model Example

```
import { Schema, model, Types } from "mongoose";

const MessageSchema = new Schema({
  from: { type: Types.ObjectId, ref: 'User',
required: true },
  to: { type: Types.ObjectId, ref: 'User',
required: true },
  text: { type: String, required: true },
  seen: { type: Boolean, default: false },
}, {
  timestamps: true
});

export const Message = model("Message",
MessageSchema);
```

---

13. Security Considerations

Security is a critical aspect of the backend to protect data, prevent attacks, and ensure trustworthiness.

- **Use HTTPS**
  Enforce HTTPS in production to encrypt all data in transit, preventing man-in-the-middle attacks.
- **JWT Authentication + Refresh Tokens**
  - Use JSON Web Tokens for stateless authentication.

    - Implement refresh tokens to maintain user sessions securely without exposing long-lived JWTs.
    - Store refresh tokens securely (e.g., httpOnly cookies).
- **Input Sanitization & Validation**
    - Sanitize all user inputs to prevent injection attacks (NoSQL injection, XSS).
    - Use validation libraries like `express-validator`, `Joi`, or `Zod` to enforce strict input schemas.
- **CORS Restrictions**
    - Restrict Cross-Origin Resource Sharing to trusted domains only.
    - Avoid setting CORS to open (`*`) in production environments.
- **Rate Limiting**
    - Implement rate limiting (e.g., using `express-rate-limit`) to mitigate brute-force attacks and API abuse.
    - Apply stricter limits on sensitive routes like `/auth/login`.
- **Additional Recommendations**
    - Secure HTTP headers with `helmet`.
    - Regularly update dependencies to patch vulnerabilities.
    - Use strong password hashing (`bcrypt`) with sufficient salt rounds.
    - Log security events for auditing.

---

14. Error Handling & Logging

Robust error handling and logging improve system reliability, debugging efficiency, and operational monitoring.

- **Centralized Error Middleware**

    - Capture and handle errors in a centralized Express middleware.
    - Return consistent error responses with meaningful messages and status codes.
- **Winston Logger**
    - Use Winston for flexible, leveled logging (info, warn, error).
    - Log to multiple transports: console, files, or remote services.
- **Log Errors & Important Events**
    - Log all application errors with stack traces.
    - Log critical events like user login, failed authentications, and important state changes.
- **Monitoring with Papertrail (or similar)**
    - Stream logs to Papertrail, Datadog, or other log aggregation platforms.
    - Set up alerts on error spikes or suspicious activity.
- **Example: Basic Error Middleware**

```
import { Request, Response, NextFunction } from
"express";

export function errorHandler(err: any, req:
Request, res: Response, next: NextFunction) {
  console.error(err);
  res.status(err.status || 500).json({
    status: "error",
    message: err.message || "Internal Server
Error",
  });
}
```

---

15. Testing Strategy

Testing is essential to ensure code correctness, reliability, and maintainability. This project employs multiple testing levels, covering units, APIs, real-time sockets, and optionally end-to-end flows.

Unit Testing

- **Tools:** Jest + ts-mockito
- **Purpose:** Test individual functions, services, and modules in isolation.
- **Examples:**
  - Authentication logic
  - Utility functions
  - Database service mocks

```
import { mock, instance, when, verify } from 'ts-
mockito';
import { AuthService } from
'../services/auth.service';

describe('AuthService', () => {
  it('should return token on valid login', async
() => {
    const authService = new AuthService();
    const token = await
authService.login('user@example.com', 'password');
    expect(token).toBeDefined();
  });
});
```

API Testing

- **Tools:** Supertest + Jest

- **Purpose:** Test REST endpoints including authentication, CRUD operations, and authorization flows.
- **Approach:**
  - Send HTTP requests to Express server.
  - Assert HTTP status codes and response payloads.

```
import request from 'supertest';
import app from '../app';

describe('POST /auth/login', () => {
  it('should respond with JWT on valid
credentials', async () => {
    const response = await request(app)
      .post('/auth/login')
      .send({ email: 'test@example.com', password:
'password123' });
    expect(response.status).toBe(200);
    expect(response.body.token).toBeDefined();
  });
});
```

WebSocket Testing

- **Tools:** Mock Socket.IO client or libraries like `socket.io-client` for tests.
- **Purpose:** Verify real-time event emission and reception, connection handling, and authentication.
- **Key Tests:**
  - Connection with valid/invalid tokens
  - Emission of custom events (`send_message`, `new_order`)
  - Proper reception by intended recipients

End-to-End (E2E) Testing (Optional)

- **Tools:** Cypress or Playwright
- **Purpose:** Test full user flows by simulating UI interactions and backend responses.
- **Scope:**
  - Login flows
  - Chat UI sending/receiving messages
  - Order placement and status updates

Summary Table

Test Level	Tools	Focus
Unit	Jest, ts-mockito	Business logic, utilities
API	Supertest, Jest	HTTP endpoints, auth flows
WebSocket	socket.io-client	Real-time event flows
E2E (Optional)	Cypress, Playwright	Full system integration

---

16. Deployment Plan

- **Dockerized build**
- Env vars in `.env`
- Platforms: Render, Railway, AWS EC2
- DB: MongoDB Atlas or Docker

---

17. Scaling Considerations

- Redis adapter for Socket.IO clustering
- Load balancer (e.g., Nginx)
- PM2 or Kubernetes
- MongoDB sharding

---

18. Future Enhancements

- Push notifications (FCM, Web Push)
- Audio/video chat
- Multi-admin dashboard
- Analytics: charts, KPIs
- Admin role management

---

19. Appendices

A. Sample API Response

```
{
  "status": "success",
  "data": {
    "token": "<JWT>"
  }
}
```

B. Socket Event Payloads

Event	Payload
send_message	{ from, to, text }
receive_message	{ from, to, text, createdAt }
user_online	{ userId }

Event	Payload
`new_product`	`{ productId, name, price }`
`order_status_update`	`{ orderId, status }`

This documentation is **ready for production**, **team onboarding**, or **open source publishing**.