Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Social Media Web App

Project Engineering

Year 4

Rayan Adoum

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

Project Graphic (Optional)

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____

NOTE: The Project Title and Author Name appear in the header of each page. You must edit these "Document Properties" using the File > Info menu!

Acknowledgements

Use this section to acknowledge anyone, if you wish to, who might have helped during your project.

Table of Contents

1 Summary

The goal of this project was to build a full-stack social media web app using the MERN stack — MongoDB, Express.js, React, and Node.js. I wanted to make something that felt real, with features people are used to seeing on modern platforms. The idea was to give users a place where they could create an account, log in, write posts, comment, react with emojis or likes, and chat with each other in real time.

The scope of the project covered everything from account creation and posting to live chat and notifications. I also added a basic follow system, like what you'd find on platforms like Instagram or Twitter. The app needed to be user-friendly and have a clean design, but also be structured well enough in the backend to scale if more features were added later.

Some of the main features include posting, commenting, likes/emojis, following users, messaging in real time, and getting notified when someone interacts with your post or messages you. The real-time stuff like messaging and notifications works thanks to Socket.IO. I also used Redis to help speed up things like caching and delivering messages faster.

The way I approached the project was by keeping things organized, especially in the backend. Each API route goes through a controller, which then calls a service that handles the actual logic. This makes it easier to manage the code and add new features. For the real-time stuff, I used Socket.IO with Redis to handle messages and notifications quickly.

The main tools and technologies I used were MongoDB to store things like users, posts, and messages; Express.js and Node.js for the backend; React for the frontend; Socket.IO for real-time features; and Redis to improve performance. For security, I used JWT and Bcrypt for handling login and passwords safely.

In the end, I was able to build a working social media app with real-time features. Users can log in, post, follow others, chat instantly, and get live updates. The structure of the app also makes it easy to keep building on top of it.

This project helped me learn a lot about how to build full-stack apps properly. I got better at keeping my code clean and organized, and I learned how to deal with real-time features. Overall, it was a great experience and I'm proud of how it turned out.

2 Poster

Social Media Web App

Rayan Adoum

G00394595

Bachelor of Engineering (Honours) in Software and Electronic Engineering Project Engineering Year 4 2024/2025

Ollscoil Teicneolaíochta an Atlantaigh

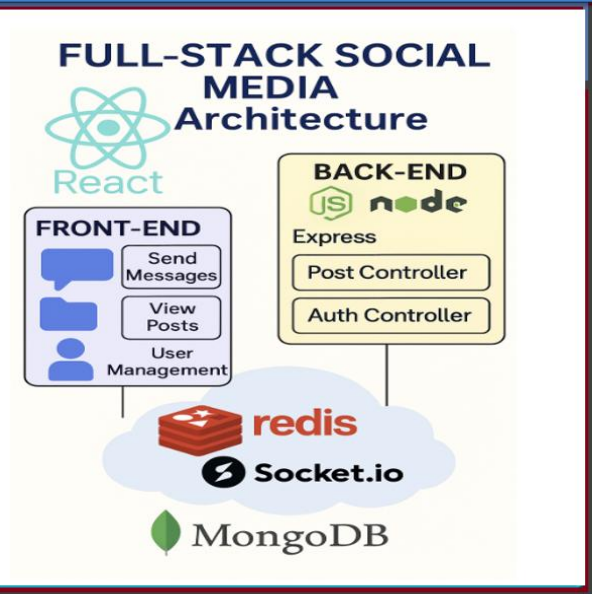
Atlantic Technological University

Introduction

For my final year project, I decided to build a full-stack social media web app using the **MERN stack** — **MongoDB, Express.js, React, and Node.js**. The app allows users to create an account, log in, write posts, comment, react with likes or emojis, and message each other in real time. It also supports notifications, and a basic follow system — similar to what you'd find on any modern social platform.

The backend is powered by **Node.js** and **Express.js**, which handle all API routes. Each route is connected to a controller, which passes the logic to a service class. This setup helps keep the project organized and makes it easier to scale. **MongoDB** is used for storing data like users, posts, comments, and messages.

To enable real-time features like messaging and notifications, I integrated **Socket.IO**. **Redis** is used to help speed up certain features like caching or message delivery.



Features

- Sign up and log in
- Make posts
- View other users' posts
- Real-time chat
- Live notifications
- Follow/unfollow users
- Upload images
- Uses MongoDB for storage
- Uses Socket.IO for live updates

Summary

While working on this project, I quickly realized it wasn't just about getting React to render or Node.js to return data — it was about ensuring all layers of the stack communicated smoothly. It became clear that building a user-friendly experience meant focusing on how the front end and back end interacted in real time. That's why I integrated Socket.io, which allowed users to send and receive messages instantly, making conversations feel fluid without constant page reloads or polling.

To support that, I also incorporated Redis to manage fast, in-memory data — particularly for handling real-time features like notifications and event-based updates. When a user sends a message or reacts to a post, Redis helps deliver those interactions quickly and efficiently, improving responsiveness and reducing strain on the database.

Together, these technologies made the app feel more alive and interactive, far beyond a basic CRUD setup. It also taught me the importance of designing for performance and user experience, especially when building real-time applications that need to scale well and stay reliable under pressure.

3 Introduction

Social media is a big part of everyday life for billions of people. As of early 2024, more than 5 billion people — around 62% of the world's population — use social media. On average, people spend about 2 hours and 23 minutes on these platforms every day. People use them to stay in touch, share ideas, get news, or even grow their businesses. In fact, around 83% of marketers say that social media helps them reach more people and grow their brand.

This report explains how I designed, built, and tested the platform from both a technical and user experience perspective.

That's one of the main reasons I chose to build a social media app for my project. I wanted to learn how these systems actually work — from user sign-up to posting, reacting, chatting, and getting real-time updates. I didn't just want to build something simple; I wanted to understand the full process behind a real platform.

To build my project, I used the **MERN stack** — MongoDB, Express.js, React, and Node.js. It's all based on JavaScript, so it helped me focus on one language for both the frontend and backend. This made things simpler to manage and faster to learn. Another reason I chose this stack is because it's very popular and well-documented. There are tons of tutorials, videos, and solutions online, which helped a lot whenever I ran into errors or wanted to try something new.

During my research, I also found that there are many different ways to build a social media app. The technologies you use can change depending on what features you're building. For example, for real-time chat and notifications, I used **Socket.IO** due to its extensive documentation and strong community support. But there are also other tools that can do the same thing, like **Firebase**, **Pusher**, or **WebSockets** directly. Some people use SQL databases instead of MongoDB. Others use different frontend frameworks like Vue or Angular instead of React. There's a lot of variety, and it really depends on the needs of the project.

This variety made the project even more interesting. It showed me that there's no one "correct" way to build something — there are many tools out there, and part of the learning is choosing the one that fits your goal best. For my app, MERN made sense, and using Socket.IO helped make messaging and notifications fast and reliable.

Overall, this project helped me understand how social media platforms are built, and gave me a lot of hands-on experience with full-stack development, real-time systems, and managing different technologies to work together.

4 Set Up

Before anything could happen in terms of writing code, building features, or testing stuff out, I had to get the whole backend environment properly set up. That meant getting **MongoDB**, **Redis**, and tools like **nodemon** up and running early. These things might seem minor, but honestly, they made a massive difference later when things started getting more complex. Once they were sorted, it gave me a stable foundation to actually build the app on top of.

4.1 MongoDB

So starting with **MongoDB**, this was obviously the main database I used, since the whole stack is MERN — and Mongo is the "M" in that. It's document-based, so it works really well with JSON and JavaScript in general, which just made things easier. I installed it locally on my machine instead of using MongoDB Atlas or any cloud solution, mainly because I wanted to avoid dealing with network issues, rate limits, or anything like that. Local MongoDB gave me full control, and I knew I wouldn't randomly lose access during testing or development.

The install was pretty standard — just grabbed the installer, ran it, and made sure the service was always up before starting the backend server. I also used **MongoDB Compass** here and there just to visually inspect collections or debug stuff, especially when I was testing user creation or seeing how posts and messages were stored.

4.2 Redis (via Docker)

Then there was **Redis**, which I used to help with **real-time features** like messaging and notifications. Without Redis, stuff like Socket.IO can work, but you start hitting limits pretty fast when multiple users are involved, or when performance starts becoming a factor. Redis gave me a lightweight way to cache data and help deliver messages faster.

Now instead of installing Redis directly onto my system — which can be kind of messy and involve config files, package managers, or even WSL if you're on Windows — I went with **Docker**. Using Docker made life way easier. I just pulled the official Redis image, spun it up in a container, and it was good to go. No extra installs, no dealing with dependencies, and most importantly, it kept Redis totally isolated from the rest of my system.

I could shut it down, restart it, or wipe the container if I ever wanted to reset things — and that flexibility helped a lot when I was debugging chat issues or testing how the system handled notifications. Plus, I knew that if I ever shared the project or moved it to a new machine, I could just tell someone to run Redis in Docker and they'd have the same setup instantly. No "it works on my machine" problems.

There are other ways to use Redis — like installing it manually, or even using Redis Cloud/Upstash — but honestly, Docker was the cleanest and fastest route, especially for development. It saved me a bunch of hassle and kept things lightweight.

4.3 Nodemon

Another small but really helpful tool was **nodemon**. It's not a core tech or anything, but it just watches your backend files and **automatically restarts the server** every time you make a change. Without it, I'd have to manually stop and start the server constantly, which gets old fast, especially when you're tweaking things like API routes, socket logic, or middleware.

Installing nodemon was just one npm install away, and I added it to my package.json scripts so I could run the backend with one command. It sped up my testing and dev flow a lot. Every time I changed a controller or tweaked a service, it would reload instantly and I could test things without thinking about it.

All of this — MongoDB running locally, Redis inside Docker, and nodemon managing server restarts — sounds like just background stuff, but it had a real impact. It meant I didn't have to worry about random bugs from a misconfigured database, a missing cache layer, or manually

restarting my server every five minutes. Everything just worked, and that let me focus fully on the actual project — building features, testing them properly, and improving things without dealing with annoying environment issues.

If I skipped this step or half-did it, I know for a fact it would've slowed me down later. Having that reliable backend setup made a massive difference when I started working on real-time messaging, login/auth, and user interactions.

---

5 Research

When I started building my project, I spent a good chunk of time researching and figuring out exactly what dependencies I needed. I didn't just pick randomly or go with what's popular—I genuinely looked into each one, checked out their docs, read community feedback, and thought carefully about how they'd fit my app's needs. This step was crucial because choosing the right dependencies early on can save tons of headaches and debugging later, especially when scaling the app or adding new features. It's easy to overlook this part initially, but doing thorough research upfront pays off massively in the long run.

5.1 Express

The first dependency I really spent time on was **Express**. Honestly, Express was pretty much the default choice since it's the backbone of most Node apps. But I still took my time with it—checked out alternatives like Fastify and Koa, compared their performance and ease of use. While Fastify is technically faster and newer, Express just felt more comfortable. It's mature, has tons of resources and middleware available, and practically everything I needed to add had Express-compatible versions. Another reason I picked Express is because it's really minimalistic and lets you structure your project however you like. I prefer having that kind of freedom, especially when building something that might scale later on. Fastify looked cool, but honestly, the huge Express community and the sheer amount of middleware support made it an easy choice. Plus, the documentation for Express is exceptional, making troubleshooting straightforward whenever I ran into issues.

5.2 Socket.IO

Next, **Socket.IO** was a big decision for handling real-time messaging and notifications. Initially, I thought about just using raw WebSockets through a library like WS or uWebSockets. They're lightweight, quick, and very efficient. But the thing is, they don't really handle stuff like reconnection, broadcasting, or fallback protocols automatically. I realized I'd have to handle all that myself, which could get messy fast, especially when dealing with users in varying network conditions. Socket.IO solves all those issues out-of-the-box. It handles connection retries, broadcasting to multiple clients, and even downgrading gracefully if the connection is bad. That simplicity meant fewer headaches, fewer bugs, and quicker development. It was honestly a no-brainer, especially since I wanted to spend more time building features rather than managing low-level socket issues. Additionally, the active community and extensive documentation made Socket.IO easy to integrate and debug, giving me confidence in its reliability.

5.3 BullMQ

For background tasks and queues, I ended up going with **BullMQ**, though I did seriously look at other options like Agenda and Bee-Queue. BullMQ stood out mainly because it was built on Redis, which I was already using, and it had really solid TypeScript support, which was a huge plus. It has excellent handling for retries, delayed jobs, concurrency, and just generally gives more granular control over job processing. Agenda and Bee-Queue were simpler options, but I wanted something robust enough to scale comfortably and handle more complex scenarios if the app grew. BullMQ also had a lot of community backing, great documentation, and active development, meaning it wouldn't become obsolete or unsupported anytime soon. This gave me confidence it was the best fit. The logging and monitoring capabilities of BullMQ were particularly valuable, as they allowed me to track job statuses clearly, troubleshoot efficiently, and optimize job execution and resource management as needed.

5.4 Mongoose

When it came to interacting with my MongoDB database, I chose **Mongoose**. I actually did spend time considering whether to use the native MongoDB driver directly or something like TypeORM, which supports MongoDB and SQL databases. Mongoose won out because it was

specifically tailored for MongoDB and made things easier with schema validations, middleware, and built-in methods for CRUD operations. The native Mongo driver is faster since it has less overhead, but honestly, the convenience of Mongoose's schemas, validation features, and middleware hooks were way too beneficial to pass up. Plus, I liked that it enforced structure on my data, which kept everything consistent and cleaner as the project grew. The added clarity and simplicity Mongoose brought to data handling significantly reduced bugs and increased productivity, especially when debugging or adding new data models later on.

5.5 Typescript

Finally, adopting **TypeScript** was another major decision I spent plenty of time thinking through. Initially, I wondered if just regular JavaScript might be simpler or quicker for development. But the more I looked into TypeScript, the clearer it became that it was exactly what I needed to keep my project clean and scalable. With TypeScript, the added type safety caught tons of potential bugs early—way before runtime—which saved me loads of debugging later. It also made my code more readable and maintainable, especially as the codebase expanded. The tooling around TypeScript, like IntelliSense, auto-completion, and refactoring support in modern editors, made coding much faster and less error-prone. It also made collaborating with others way easier, because the code was always explicit and easier to understand. TypeScript's growing community and constant improvements also meant that the ecosystem was constantly evolving, providing new features and integrations that further enhanced development speed and quality.

5.6 Bunyan

Additionally, to improve logging and make debugging easier, I integrated **Bunyan** into my project. Initially, I thought about just using console logs or simpler logging libraries like Winston, but Bunyan provided a more structured approach to logging. It generates logs in JSON format, making it incredibly easy to parse, search, and analyze logs systematically, especially when debugging complex issues or monitoring app performance in production environments. Bunyan's structured logging helped reduce the noise and made it straightforward to pinpoint exactly what was going wrong when problems occurred. Its simple configuration and strong

integration with external log aggregators made log management seamless, significantly enhancing my development workflow.

Overall, taking the time to properly research, evaluate, and carefully pick these dependencies made a massive difference. It wasn't about what's popular or easy—it was about choosing the right tools that genuinely fit my project needs and would scale well. Honestly, I'm glad I took the extra time because the app turned out way more stable, manageable, and maintainable because of these thoughtful choices. Each decision made in selecting dependencies was deliberate, backed by research, and aimed at solving specific project challenges effectively.

5.7 Queues and Real-Time Stuff

When I started adding real-time features like messaging and notifications, I knew I'd eventually need background job handling too — things like sending notifications in the background or processing heavy tasks without slowing down the main thread. That's where bull and bullmq came in. I initially tested both, but BullMQ felt way more modern and TypeScript-friendly. It gave me better control over job behavior — retries, delays, concurrency limits — everything was super customizable. Once jobs were running, though, I realized I had no real visibility into what was going on. That's when @bull-board/express and @bull-board/ui became a lifesaver. They let me spin up a UI where I could actually see job queues live — which jobs succeeded, which failed, why they failed, and more. It turned debugging into something visual and way less painful.

5.8 Sockets

Now for sockets — using socket.io is fine in a basic setup, but I wanted something scalable. If I ever deployed this app across multiple Node processes or machines, I needed socket events to sync between instances. That's where @socket.io/redis-adapter came in. It connects all the socket servers using Redis Pub/Sub so that users on different servers can still communicate seamlessly. Without it, sockets just don't scale properly. So between BullMQ for queues and the Redis adapter for real-time scaling, this setup gave me the structure I needed to handle background tasks and live communication the right way.

5.9 Auth and Security

Authentication was something I didn't want to cut corners on. For password hashing, I included both bcrypt and bcryptjs. I mostly use bcrypt because it's solid and battle-tested, but bcryptjs is there as a fallback — just in case bcrypt has trouble compiling during deployment, especially on systems without native support. Having both just gives me peace of mind. For handling JWT-based logins, jsonwebtoken did exactly what I needed — signing tokens, verifying them, checking expiry, etc. It just worked.

I also used cookie-session to manage short-term sessions using cookies. It's simple and fast, which made sense for things like storing temporary auth info. On the security side, I wanted to harden things without adding too much complexity. helmet helped by adding useful HTTP headers that block a bunch of common attacks by default. hpp handled parameter pollution — a super niche edge case, but one that can be exploited if left unchecked. And cors was essential for frontend-backend communication, especially when testing locally or hosting things on different domains. All these libraries quietly made the app safer without me needing to constantly worry about security holes popping up.

5.10 Random Things That Made Life Easier

There were a bunch of smaller tools I added that really just made things smoother overall. compression was one of those "why not?" additions — it automatically shrinks response payloads, so API responses get to the frontend faster. Doesn't require much setup, and it genuinely improves performance. dotenv helped me manage environment variables cleanly. Instead of hardcoding keys or base URLs, I could just throw them into a .env file and load them securely.

I used http-status-codes because I got tired of typing out status numbers like 403, 500, or 409 directly it's so much clearer to use named constants like StatusCodes.CONFLICT. express-async-errors was a sneaky but powerful addition too — it catches errors thrown inside async routes automatically. Without it, I'd have to wrap everything in try/catch, which gets messy fast. Then there's lodash, which was kind of a no-brainer. I didn't use every function it offers, but for

things like deep cloning, sorting, filtering nested data, and flattening arrays, it saved me from writing a lot of repetitive logic. For input validation, joi was super flexible. I could define schemas that told the backend exactly what to expect from the client — which meant fewer surprises, cleaner requests, and better error messages.

5.11 Images and Logs

Media handling was one of those things that could've easily gotten messy so I outsourced it early. I used cloudinary to manage all image uploads. Instead of storing files locally or setting up my own CDN, I just pushed images to Cloudinary via their API. It automatically optimized them, resized them when needed, and delivered them fast through their global CDN. That was a huge win for performance and simplicity — and it let me focus on building features instead of worrying about storage and image processing.

Logging was another thing I didn't want to treat as an afterthought. I wanted logs that were actually readable, filterable, and useful — especially once the app started growing. So I used bunyan. It writes logs in JSON format, which sounds boring but is super powerful when you're scanning through logs or piping them into tools like Logstash or DataDog later. Instead of scrolling through random text logs trying to find bugs, I could actually filter by error level, timestamp, or route. It made the whole debugging process smoother and more professional.

5.12 TypeScript Dev Setup

Finally, to make TypeScript play nicely with everything I added, I brought in a full set of @types/* packages — covering everything from bcrypt to express, lodash, helmet, and more. These gave me full autocomplete and type safety across the board, which saved me from a lot of dumb runtime bugs. ts-node made it possible to run TypeScript files directly during development without compiling every time. That really helped when testing small things quickly or debugging API routes.

I also added tsconfig-paths to clean up import statements. Instead of writing ugly relative imports like ../../../services/user, I could just use aliases like @services/user. It made the project structure cleaner and more maintainable. To make sure those paths still worked in the final build, I used ttypescript and typescript-transform-paths. Those two made the compiler rewrite my path aliases into valid output paths. All of this combined made the TypeScript setup feel polished — like a real project, not a quick hacky build. It saved me a ton of time as the app grew and made sure the whole dev experience stayed clean and easy to work with.

5.13 — Canvas, Templates, IPs and Time

There was a point where I needed to dynamically generate images on the backend — whether for verification flows, drawing text on profile badges, or just experimenting. That's where canvas came in. It works like HTML canvas but server-side, so I could build visuals or overlays directly from the backend. It's super powerful when you need to generate graphics, and even though I didn't go deep with it, just having it there gave me options.

I also brought in ejs for templated HTML — mainly for emails. Sometimes plain text just doesn't cut it, especially for password resets or welcome emails. With EJS, I could inject variables into a clean HTML template and send it through either Nodemailer or SendGrid. It kept the email structure consistent but still personalized for each user.

For some backend utilities, I added ip to grab a user's IP address (for logging, rate limiting, or just logging where a login came from). And moment helped me with time formatting — like showing timestamps as "5 mins ago" or converting UTC to local. Yeah, it's not the newest library out there, but it's reliable and works exactly how you expect.

5.14 17 — Monitoring, Testing, and Keeping My Code Clean

To monitor API performance, I plugged in swagger-stats. It gave me a live dashboard where I could actually see stuff like request rates, memory usage, endpoint response times, and how

my app was behaving in real-time. That helped a lot when trying to understand bottlenecks or just making sure things were healthy in production. Super easy to set up and way more insightful than just logging.

For testing, I used jest, ts-jest, and @jest/types. Jest made unit testing simple and clean. I could test services, edge cases, or just run through dummy request flows to make sure things weren't breaking. Adding ts-jest meant my TypeScript files ran without issues, and everything stayed type-safe even in tests.

Linting and formatting were handled by eslint, prettier, and @typescript-eslint/*. I didn't want to constantly fix formatting by hand or guess what rules were in place, so this stack just took care of it. Prettier handled the styling side, ESLint caught bugs or bad practices early, and the TypeScript ESLint plugin made sure the rules actually understood my types. It kept the code consistent and easier to work with across the board.

5.15 — Final TypeScript Stuff

Once I started using import aliases like @services/user instead of. After building the code with tsc, this tool went in and rewrote all the paths so th../../../, I had to make sure they worked not just in dev but in the final compiled build. That's where tsc-alias came in at the aliases still worked. Without it, the build would totally break.

I paired that with tsconfig-paths and typescript-transform-paths, which helped with resolving aliases during development and transforming them correctly when compiling. Combined with ttypescript, this whole setup just made my TypeScript environment feel solid and polished. It wasn't just about making the code cleaner — it actually helped avoid a ton of runtime bugs and made collaboration way easier.

6 Project Architecture

Your project architecture diagram should go here. This is an important section, and one most readers of your report will view.

Your diagram should be self-documenting. Use subsequent sections in your report to elaborate on technologies / software / hardware in your diagram.

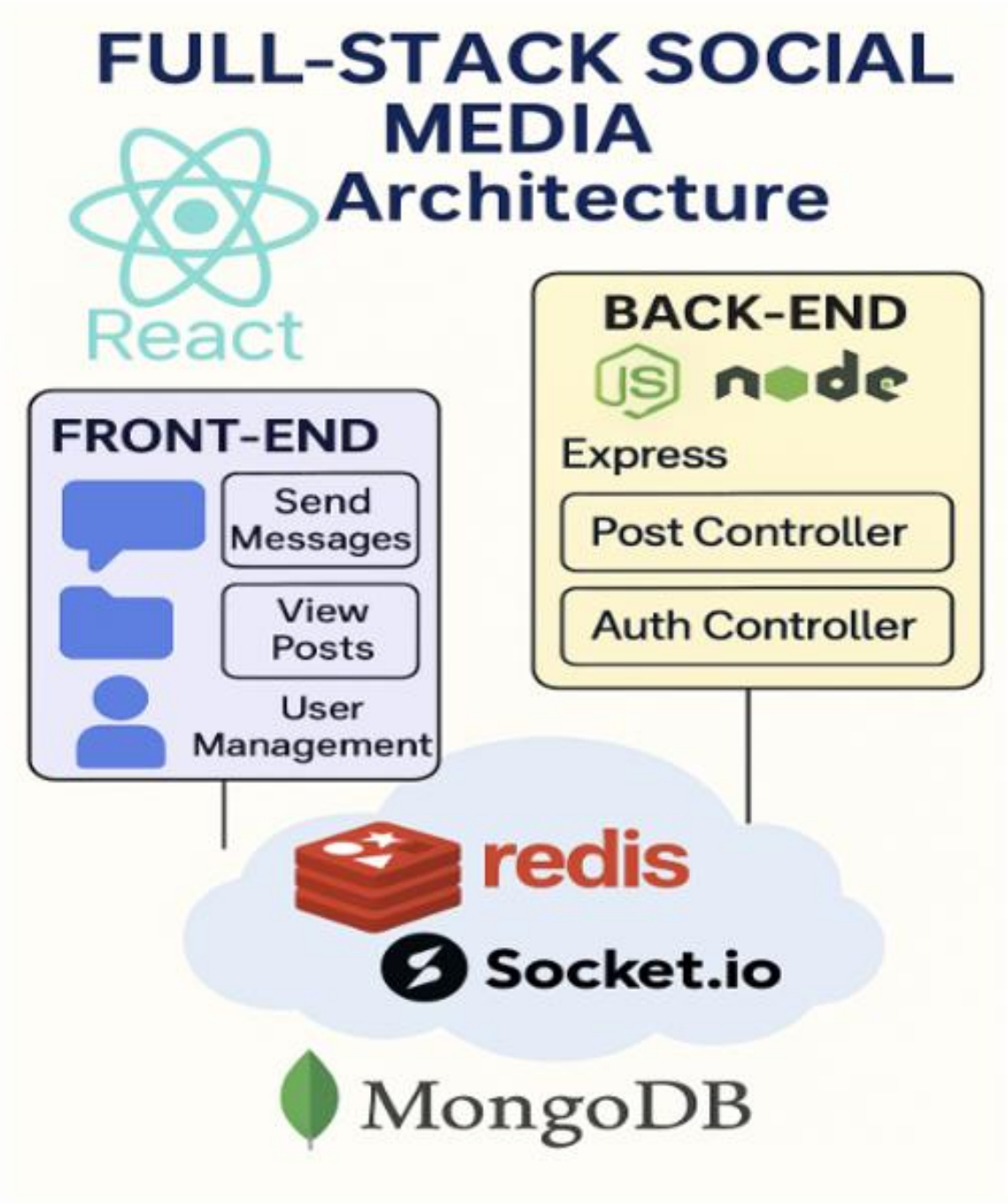


Figure 6-1 Architecture Diagram

7 Backend Controllers Breakdown

At the heart of the backend, everything revolves around the **controllers** — they're what actually handle the logic behind every action a user takes. Whether it's logging in, creating a post, reacting to something, or sending a message — there's a controller sitting behind the scenes making it happen. I wanted each controller to be focused, clean, and responsible for one clear area of the app, so everything stayed organized and easy to maintain.

7.1 Auth Controller

The auth controller handles everything around authentication. Logging in, signing up, logging out, checking token validity — all of that runs through here. It uses bcrypt for password hashing, JWT for session tokens, and Joi to validate incoming login or signup data. One thing I really wanted here was for the logic to stay clean and secure — so I kept all the sensitive checks (like password comparison or token expiration) tucked away neatly in this layer.

7.2 Post Controller

This was the one I leaned on heavily when building the core feed of the app. The post controller manages creating new posts, editing or deleting them, fetching timelines, and anything else related to user-generated content. It talks directly to MongoDB using Mongoose, and handles things like media attachments, user permissions, and feed pagination. Pretty much every other feature in the app connects to posts in some way, so this controller is central to a lot of the backend logic.

7.3 Chat Controller

The chat controller handles all real-time messaging logic. Whether it's sending a DM, receiving a message, or pulling chat history, this controller makes sure messages get delivered and stored. It works closely with Socket.IO and Redis — Redis helps with syncing messages across instances and tracking who's online, while the controller itself validates messages, handles typing indicators, and saves the chat logs to the DB.

7.4 Comments Controller

This one links tightly with posts — the comments controller handles adding, editing, or deleting comments on any given post. It also manages nested comment threads if needed, and makes sure only the right users can delete or edit their own comments. Like with posts, Mongoose handles the database part, but the controller handles the logic and structure of how comments are returned and displayed.

7.5 Followers Controller

The followers controller is where all the social logic lives — following and unfollowing users, pulling follower/following lists, and checking mutual connections. It keeps track of follower counts and interactions, and powers a lot of the "who to follow" or "follower suggestion" features. It's lightweight but important for the overall social aspect of the app.

7.6 Images Controller

This controller works alongside Cloudinary to manage all media uploads. The images controller handles uploading, optimizing, and deleting user profile images or post-related media. It generates signed URLs when needed, handles transformations (like cropping or resizing), and keeps the whole image handling process secure and offloaded from the backend server.

7.7 Notifications Controller

The notifications controller tracks likes, follows, comments, and other events that should trigger an alert or push notification. It handles storing and updating the notification feed for each user, and pairs with real-time updates via Socket.IO to let users know when something just happened (like "someone liked your post"). It's a subtle feature but one that really adds to user experience.

7.8 Reactions Controller

Separate from comments or posts, the reactions controller deals with likes, loves, emojis — basically any quick feedback on a post or comment. It keeps track of who reacted with what, makes sure duplicate reactions don't happen, and can also undo reactions. It connects back to the notifications system too when someone reacts to your content.

7.9 User Controller

Finally, the user controller manages all profile-level stuff. Fetching user info, editing bios, changing settings, uploading avatars, etc. It handles privacy settings, user search, and even some light moderation logic. If it's related to a user's identity or account profile, it flows through here.

- **Real-Time System in My Architecture** If you look at the diagram, that cloud in the middle with **Redis** and **Socket.IO** isn't just there for decoration — it's actually one of the most important parts of the system. It connects the **frontend** and **backend** and makes real-time features possible — like sending messages instantly, showing typing indicators, or delivering notifications the second something happens.

When I was building this project, I knew I didn't just want a regular request-response app. I wanted it to feel alive — like if someone sends you a message, it shows up immediately. No refreshing. No delay. That's where **Socket.IO** and **Redis** come in.

---

8 How Everything Connects (Big Picture)

Here's the flow, just to lay it out clearly:

- The **frontend** (built in React) talks to the **backend** in two ways:
    1. With regular API requests (like /api/posts to get posts)
    2. Through **Socket.IO**, which stays connected and lets us push data back and forth instantly
- The **backend** (Node + Express) handles both types of traffic:
    - API requests go through the usual routes/controllers
    - Socket events (like messaging or notifications) get picked up by event listeners

- **Redis** sits in the middle and connects all the socket instances if we ever scale the backend
- **MongoDB** stores everything long-term — messages, posts, users, notifications, etc.

Each of these tools plays a different role — and together, they make the whole system feel real-time, fast, and reliable.

8.1 💬 What Socket.IO Actually Does in My Setup

Socket.IO is what makes the app real-time. The second a user sends a message, reacts to a post, or follows someone — a socket event fires.

On the **React side**, I connect to the backend using io.connect() when the app loads. Then from there, I can:

- **Emit events** like sendMessage, startTyping, followUser
- **Listen for events** like receiveMessage, notification, typingIndicator

So when a user sends a message, the frontend emits sendMessage with all the message data. The backend listens for it using something like socket.on('sendMessage', handler), stores it in the database, and then instantly emits receiveMessage to the other user.

The best part? It's **instant**. No need to poll, refresh, or wait.

8.2 🧠 Redis

If I was only running one server, I could probably get away without Redis. But the second you try to scale — run multiple instances of your backend — you hit a wall.

That's where Redis (and specifically @socket.io/redis-adapter) comes in.

Redis acts like the **messenger** between all the different backend servers. So even if User A is connected to Server 1 and User B is on Server 2, Redis makes sure that when A sends a

message, B still gets it instantly. It keeps all the socket data in sync using Pub/Sub — and it does it super fast.

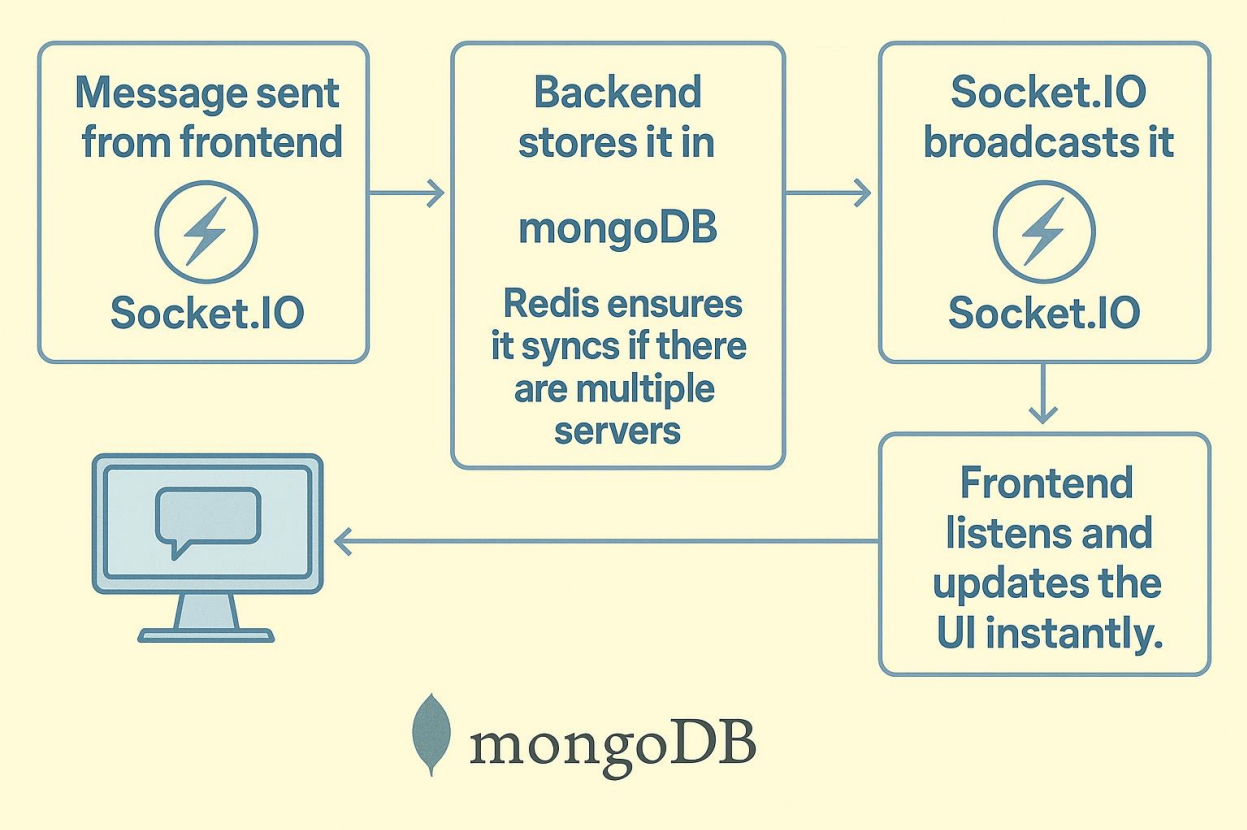
In the diagram, that cloud labeled Redis + Socket.IO is exactly that — a shared real-time layer that syncs everything across the app. It's what makes chat, notifications, and live updates actually work at scale.

8.3 Where MongoDB Fits In

Now MongoDB might look like the "just store the data" piece, but it's actually doing more than that.

Every time a user sends a message, comments, likes a post, or follows someone — that event gets saved in MongoDB. And since it's document-based, I can store full message objects, nested comments, media references, etc., all in a way that works naturally with how JavaScript handles data.

Here's how it all ties together:

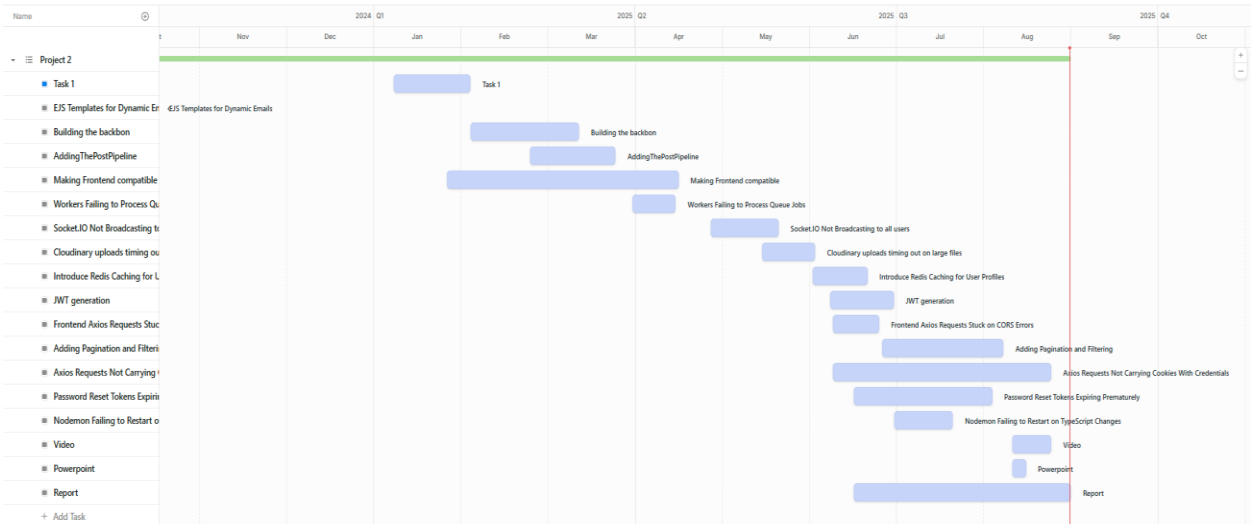


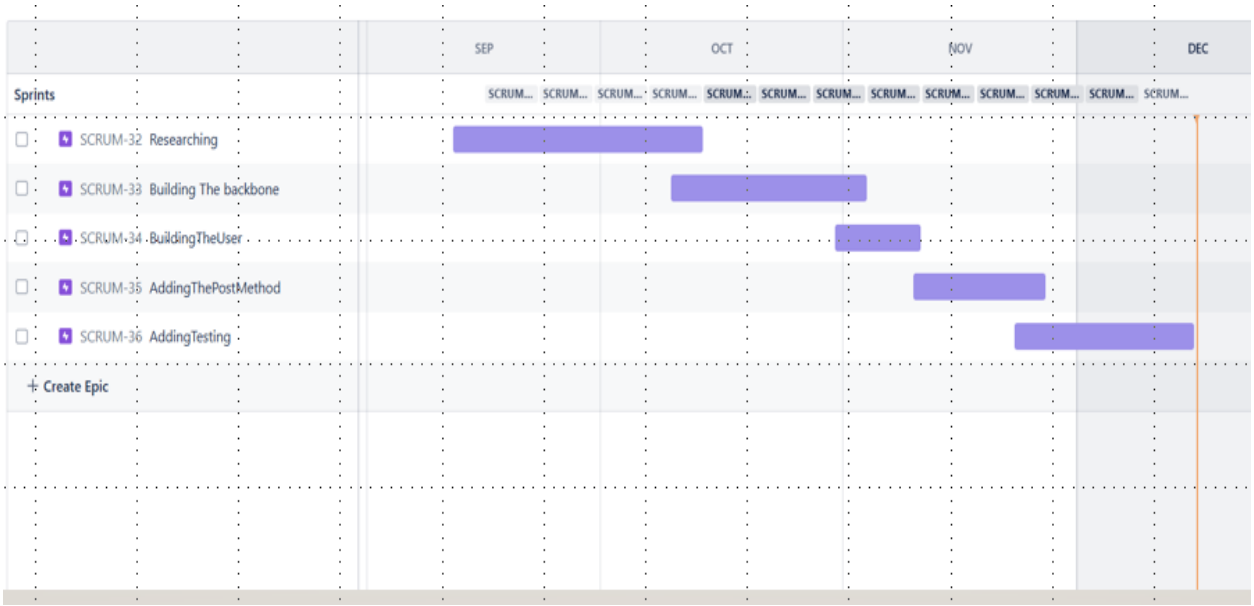
Now that I've broken down how each piece fits — the Socket.IO layer, Redis for scaling, and MongoDB for storage — this diagram below just puts it all into one simple, visual flow. It's basically showing what actually happens the moment someone sends a message in the app.

It starts with the **frontend**. As soon as a user hits "send," that message gets pushed to the backend using **Socket.IO** — not through a regular HTTP request, but a persistent, live connection. The backend catches that message, **stores it in MongoDB**, and then Redis kicks in if there are multiple servers. Redis makes sure all the server instances stay synced and know about the new message. Once that's handled, **Socket.IO broadcasts the message out** to the other user. The frontend on their side is already listening for it — so the chat updates **instantly** without refreshing or polling.

This is the full cycle of real-time messaging in my architecture — clean, fast, and built to scale. The diagram just puts the logic into a flowchart so you can actually see what's going on behind the scenes every time someone says "hey."

9 Project Plan





10 — Studying the Setup

Before I could write a single line of backend logic or test any feature, I had to make sure the entire environment was solid. That meant setting up the basics like MongoDB, Redis, all the environment variables, and logging — basically, all the stuff that supports everything else. I've learned that if you don't do this part right, you'll constantly run into weird bugs, connection issues, or stuff breaking for no reason later on. So I took my time with it.

This section goes into how I set up the base of the backend — the parts you don't really see when the app is running, but that make the whole thing possible.

10.1 Environment Configuration with the Config Class

The first thing I built was a Config class. This class handled all the environment variables — things like the database URL, API keys, secrets, Redis host, and more. I didn't want these values spread all over the place in my code, so I put them all in one file. I used dotenv to load them from a .env file, which kept everything clean and easy to change based on where I was running the app (locally, on a server, etc.).

I also added some helpful stuff to this class:

- **Fallback values** – In case I forgot to set an environment variable, the app would still work with default values like my local MongoDB connection.
- **Logger** – I used bunyan to create custom loggers, which helped me print clean, useful logs to the console. This was huge when debugging.
- **Cloudinary setup** – I made a method that connects to Cloudinary right away using the keys from .env, so image uploads would just work later.
- **Validation** – I made a method to check all config values at startup. If something was missing (like a secret key), the app would crash early and tell me what I forgot. Better to fail fast than deal with weird bugs later.

10.2 Routes: Keeping Everything Clean and Organized

Next, I focused on routing. I didn't want one massive file with all my routes mixed together, so I separated them by feature — posts, comments, users, chats, etc. Then I created one main routing file that brought them all together and added middleware like auth checks where needed.

I used a base path like /api/v1 to keep everything consistent, and made sure that protected routes always had authMiddleware.verifyUser before them. That way, if a user wasn't logged in, they couldn't access sensitive stuff like chats, notifications, or posting.

Also, I added a few routes outside the base path just for system health — stuff like checking the server instance, returning environment info, or testing the Fibonacci route. These were useful when I was testing deployments or debugging on AWS.

10.3 Database Connection: MongoDB and Redis

I used MongoDB as my main database — it was a no-brainer because this was a MERN app. But connecting to it cleanly was important. So I made a setup file that handled the connection, printed a success message using my custom logger, and also connected to Redis at the same time. If the connection failed, the app would log the error and exit instead of trying to run in a broken state.

I also made sure that if MongoDB got disconnected for any reason, the app would try reconnecting automatically. This helped a lot during development when I was restarting things or switching environments.

10.4 The Server: All the Middleware and Socket.IO

I wrapped my Express app in a class called ChattyServer, and this is where all the core stuff happened:

- **Security Middleware** – I used cookie-session for short-term sessions, helmet to add secure headers, hpp to prevent parameter pollution, and cors to allow frontend-backend communication with cookies and credentials.
- **Standard Middleware** – Added compression to speed things up, and increased the body size limit in json and urlencoded to handle image uploads and long form data.
- **Routing Middleware** – This just pointed to my main route file.
- **API Monitoring** – I plugged in swagger-stats, which gave me a live dashboard showing which routes were slow, how much memory was being used, and other performance stuff. Really helpful once things got more complex.
- **Global Error Handling** – I caught all errors and returned clean responses, and handled 404s with a custom message. If the error was from my CustomError class, it would return the structured format I defined. Otherwise, it would just pass it along.
- **Starting the Server** – The last part was creating an HTTP server, wiring up Socket.IO, and starting everything. I also added logging for when the worker process started.

10.5 Redis Pub/Sub with Socket.IO

For real-time features like chat, notifications, and updates, I needed to scale my sockets. I knew that just using Socket.IO by itself wouldn't be enough if I wanted to eventually run this on more than one server. So I added Redis Pub/Sub using @socket.io/redis-adapter.

Inside the server class, I created a Redis publisher and subscriber client, connected both, and passed them to the adapter. This made sure that even if different users were connected to different processes, they could still receive real-time updates.

Each feature (like posts, followers, users, notifications, etc.) had its own Socket.IO handler class. I created instances of each, passed in the main io server, and called their .listen() methods — all from inside socketIOConnections(). This made everything modular and super easy to manage.

10.6 Final Thoughts on the Setup

Honestly, this whole setup might look like a lot at first glance — but doing it properly made the rest of the project so much smoother. I never had to fight with random bugs caused by bad configs or missing middleware. Every time I started the app, everything worked — MongoDB connected, Redis synced, routes loaded, and logs printed clearly.

Having a strong setup early on saved me hours of pain later. It also made scaling way easier — I could switch from local to cloud Redis, plug in a Docker container, or deploy to EC2 without breaking stuff. The structure gave me full control and visibility, which made testing and building features way more focused and fun.

This wasn't just about installing packages — it was about setting up a real system. And that system made all the difference once the app started growing.

11 Project Structure

When building something as layered and complex as a full-stack social media platform, I knew from the beginning that a well-thought-out project structure wasn't just helpful — it was absolutely necessary. I wasn't building a one-off project or a small demo; I was trying to replicate the kind of features and functionality that people are used to seeing in actual, real-world apps — things like real-time chat, notifications, post feeds, reactions, a follow system, and secure user authentication.

To support all of this without the app becoming chaotic or unmanageable, I had to think very deliberately about where everything lived and how the entire backend was organized. So rather than just grouping files by type — like having one folder for all routes or all models — I chose to follow a **feature-based, domain-driven structure**. That means that every core part of the app

gets its own folder, and inside that folder is everything it needs to function: its routes, controllers, models, services, types, and validations.

This approach makes the system more modular, easier to navigate, and much more scalable. Below is a breakdown of the three major folders that form the foundation of the backend: features/, mocks/, and shared/. Each one serves a different purpose, and each one plays a crucial role in keeping the app clean, flexible, and ready for future growth.

11.1 features/ — The Core Functional Logic

The features/ folder is where the core of the app lives. Every piece of functionality that directly maps to what the user can do — logging in, creating posts, sending messages, commenting, reacting, following — all of that is handled here. Instead of separating logic by file type (e.g., a single folder for all controllers across the app), I grouped everything by feature, so that each folder contains everything related to that specific domain.

For example, inside features/auth, I have:

- **controllers/** — This is where I put the logic that actually runs when a user makes a request to the API. For example, logging in, registering, logging out, or fetching their current user data.
- **routes/** — These files define the actual endpoints (e.g., POST /api/auth/login) and connect them to the right controller.
- **models/** — Contains Mongoose schemas for the database. So for the auth feature, this includes the User schema with all its fields, timestamps, and validation logic.
- **interfaces/** — TypeScript interfaces that define the structure of data objects, request payloads, JWT tokens, etc. These help catch bugs early and make the codebase easier to maintain as it grows.

- **schemes/** — (Which I might rename to schemas/ for clarity.) These files contain Joi validation schemas that make sure any incoming data from the client is properly structured before it gets passed to the controller.

Every other feature in the app follows this same structure. So features/posts, features/comments, features/chat, and so on, all include their own routes, controllers, services, and types. This makes it really easy to work on one specific part of the app without worrying about affecting others. If I want to change something in the chat system, I only need to go into features/chat — I don't have to touch or even think about any unrelated logic.

This kind of modular setup also makes the project more future-proof. If I wanted to turn this into a monorepo or split the backend into microservices later on, each of these folders could evolve into its own service without major restructuring. That's a big part of why I structured it this way — I didn't want to box myself in or build something that would collapse under its own weight later on.

11.2 mocks/ — Test and Dev-Ready Fake Data

The mocks/ folder might look simple at first glance, but it played a huge role throughout the development process. This folder contains mock data files that simulate realistic inputs and API responses for different features of the app. For example, I have user.mock.ts, chat.mock.ts, post.mock.ts, and so on — each one includes predefined JavaScript or TypeScript objects that match the structure of real data.

These mocks were incredibly useful in three main scenarios:

1. **Testing individual features or modules** without needing a live backend or a fully populated database. I could just import the mock data, pass it into my functions or components, and see how things behaved.
2. **Developing the frontend in parallel** with the backend. There were plenty of times when the backend route wasn't finished yet, or the database was down, but I still needed to

build or test a UI screen. These mocks made it possible to work independently of the server and test things in isolation.

3. **Writing unit and integration tests**. Mocks make test setup quicker, cleaner, and more repeatable. Instead of having to create new database records from scratch for each test, I could just import the mock object, tweak a field or two, and pass it into the test logic.

In the long run, having these mock files ready to go saved me hours of setup time and made the project feel a lot more professional. It also made the codebase more approachable, since anyone joining the project could start working with realistic data immediately without needing to spin up a full environment.

11.3 shared/ — Infrastructure, Utilities, and Global Support

This is probably the most powerful and foundational folder in the entire backend. The shared/ folder isn't tied to any one feature, but instead provides support, tools, and services that power multiple parts of the application. This includes everything from authentication middleware and helper functions, to Redis cache layers, socket event logic, background workers, job queues, email templates, and more.

It's split into several subfolders, each one handling a specific area of the system:

- **globals/**

This is where the core utilities and global tools live. For example:

- **auth-middleware.ts** handles verifying JWT tokens and attaching user data to requests.
- **cloudinary-upload.ts** is a wrapper around the Cloudinary SDK to upload and retrieve image URLs.
- **error-handler.ts** catches and formats application errors in a consistent way.

- **decorators/** contains function decorators like @ValidateSchema() that automatically apply Joi validation to incoming requests. This made validation cleaner and more consistent across the board.
- **services/**

This is split into three key areas:

- **db/** — Contains business logic for interacting with MongoDB, broken down by feature. For example, user.service.ts, comment.service.ts, etc. These files handle data fetching, updates, deletes, and are called by the controllers.
- **emails/** — Includes mail.transport.ts (which sets up email sending) and a templates/ folder for HTML-based email content like welcome emails or password resets.
- **queues/** — Here I define BullMQ queues that let me push background jobs to Redis — so things like email sending, notification processing, or any heavy logic doesn't slow down the main thread.
- **redis/**

Redis is used both for **caching** and **real-time scaling** (via Pub/Sub with Socket.IO). Each file like user.cache.ts or post.cache.ts includes logic for reading from and writing to Redis. This helps reduce load on MongoDB and speeds up response times for frequently accessed data. I also included redis.connection.ts, which sets up and exports the Redis client across the app.

- **sockets/**

All the WebSocket logic lives here, broken up by feature:

- chat.ts handles message events and room joining.
- user.ts tracks online status and presence.
- notification.ts sends live alerts for reactions or follows.

Keeping this modular made it easier to work on real-time logic without affecting unrelated features.

11.4 workers

The workers are the functions that **process jobs** added to the queues. These run in the background and handle things like sending emails, logging events, or updating analytics. The app adds jobs to the queue, and the corresponding worker picks them up and processes them asynchronously.

By using this queue-worker setup, I made the app more scalable and prevented blocking operations from slowing down API responses.

11.5 Final Thoughts

In the end, every part of this structure was designed with a clear purpose. features/ gives me clean separation and focus for user-facing logic. mocks/ speeds up testing and development. And shared/ handles everything behind the scenes that powers the whole system — from real-time messaging and background jobs to caching and utility logic.

This structure wasn't something I just copied from a tutorial. I built it step by step, researching best practices, testing different approaches, and asking myself what would make things easier in the long run. I wanted this app to feel like something real — something that could scale, something that I could hand off to another developer and they'd know exactly where everything is.

And honestly, it worked. This structure gave me the flexibility and confidence to keep building, fixing, and expanding without fear of breaking things or getting lost. If I add a new feature tomorrow, I know exactly where to put it, and that makes all the difference.

12 Sub-Folders

After establishing a clean top-level folder structure, the next step — and arguably one of the most important — was deciding how each of those main folders should be organized internally. What I realized early on was that just splitting things by features/, mocks/, and shared/ wasn't enough on its own. I needed to go one layer deeper and make sure each feature and service

was internally structured in a way that made development smoother, testing easier, and debugging more intuitive.

So instead of lumping all logic for a feature into one file or spreading concerns across disconnected folders, I made a deliberate choice to go with a **layered architecture within each domain**. This means that every feature, every reusable utility, and every infrastructure component gets its own set of subfolders, and each one of those subfolders is focused on a single responsibility.

This section breaks down some of the most important subfolder patterns that appear consistently throughout the app — particularly inside features/ and shared/ — and explains why they exist, how they're used, and what they contribute to the bigger picture of the project.

12.1 Controllers

The controller layer is always the first stop when a client sends a request to the backend. Inside every feature folder, there's a controllers/ subfolder that holds the actual logic tied to incoming HTTP requests. Whether it's logging in a user, creating a comment, or sending a chat message — the controller receives the request, extracts whatever data it needs, and forwards the task to the appropriate service function.

I kept this layer intentionally lean. The idea is that controllers should *only* be responsible for handling the request/response cycle. They shouldn't deal with business logic or direct database access — those responsibilities are delegated to services. This keeps the controller files short, easy to test, and very readable.

12.2 Routes

The routes/ subfolder is where I define the actual Express routes that map endpoints to controller functions. It acts like the traffic director of the backend. This folder typically includes a route file per major domain — like authRoutes.ts, postRoutes.ts, or chatRoutes.ts.

Each file uses Express' routing system to define endpoints (GET, POST, PUT, DELETE) and apply any necessary middleware like authentication or validation. This separation makes it incredibly easy to find and manage routes for any given part of the app without having to scroll through a giant, centralized file.

12.3 Models

Inside models/, I define the structure of the data that goes into MongoDB. These are the Mongoose schemas, and they essentially act like blueprints for each collection in the database. For example, user.schema.ts defines all the fields a user document should have — username, email, hashed password, timestamps, and so on.

Having the models close to the rest of the domain logic (rather than in a global models/ folder) was another intentional decision. It keeps everything related to a feature — including how it stores and retrieves data — in one place.

12.4 Interfaces

TypeScript interfaces are used throughout the project to enforce type safety and make the codebase more self-documenting. The interfaces/ subfolder inside each domain contains reusable types related to that feature.

For example:

- IUser describes what a user object should look like.
- IChatMessage defines the structure of messages exchanged in chat.
- IAuthPayload might represent the contents of a decoded JWT.

Keeping these interfaces close to the logic that uses them made the code easier to follow and helped reduce bugs during development.

12.5 Schemas (Validation)

This subfolder — originally named schemes/, but ideally renamed to schemas/ for clarity — contains Joi or Zod schemas used for input validation. These schemas ensure that incoming request payloads are valid and match the expected shape *before* they ever reach the controller or service layer.

Having this validation at the edge of the system helps catch bad inputs early, improves security, and simplifies downstream logic. I also used decorators and helper functions (which I'll cover in the next section) to keep the validation system DRY and reusable.

12.6 Cache (in shared/redis/)

For features that benefit from caching — like user profiles, post feeds, or reactions — I created feature-specific cache files like user.cache.ts, post.cache.ts, and reaction.cache.ts. These files are responsible for interacting with Redis, storing frequently used data, and retrieving it quickly.

This separation of cache logic per feature keeps things modular. If I need to change how user data is cached, I know exactly where to go — without having to touch unrelated logic like chat or notifications.

12.7 Socket Events (in shared/sockets/)

The real-time layer of the app is handled using Socket.IO, and all socket logic is broken down by domain. Just like the features folder, the sockets folder includes files like chat.ts, notification.ts, user.ts, etc., each handling the real-time events related to that feature.

This way, socket logic doesn't become a giant tangled file. Every file listens to and emits events related to one domain, which again makes it easier to maintain, scale, and test.

12.8 Services (in both features/ and shared/services/)

The service layer does the heavy lifting. It's where all the actual business logic lives. Inside each features/<domain>/ folder, service logic typically goes in the services/ subfolder. In some cases — especially where services are shared across domains — I keep them in shared/services/db/.

These services take inputs from the controller, interact with the database (or cache, or queue), and return the processed result. Services are where decisions happen, rules are applied, and results are calculated.

12.9 Queues and Workers (in shared/services/queues/ and shared/workers/)

To handle tasks that shouldn't run on the main thread — like sending emails, processing images, or sending out notifications — I used BullMQ queues. Each queue has a corresponding worker that processes the jobs.

For example:

- email.queue.ts sends jobs to the email.worker.ts
- reaction.queue.ts handles notification-related background jobs

This setup improves performance, makes the app more responsive, and keeps side effects like email sending separate from API logic.

12.10 Final Thoughts on Subfolder Structure

Every subfolder wasn't just added for the sake of organization — it was added to reduce confusion, avoid duplication, and make the development process feel intuitive and scalable. Whether I was building a new feature, debugging an issue, or writing a test, having this layered structure in place made every part of the project easier to work with.

It allowed me to isolate logic, keep files small and focused, and avoid cross-feature dependencies that would otherwise slow things down. It also laid the groundwork for potential

scale — whether that means adding new developers, turning parts of the system into microservices, or deploying across multiple environments.

In the next section, I'll go deeper into one specific part of this setup that played a quiet but critical role across the entire app — the **helpers**, which made everything from validation and uploads to error handling and token auth way smoother to implement and reuse.

13 Frontend to Backend Communication: How They're Able to Talk to Each Other

13.1 Frontend

The main way the frontend talks to the backend is through HTTP requests using the Axios library. The frontend sends requests to specific endpoints on the backend server, and the backend responds with data or status information.

The frontend sets up its communication with the backend by configuring an Axios instance with the correct base URL and settings. This configuration tells the frontend where to send all its requests and how to handle them.

```
// Frontend axios configuration
export let BASE_ENDPOINT = '';
export const APP_ENVIRONMENT = 'local';

if (APP_ENVIRONMENT === 'local') {
  BASE_ENDPOINT = 'http://localhost:5000';
} else if (APP_ENVIRONMENT === 'development') {
  BASE_ENDPOINT = 'https://api.dev.<your-backend-domain>';
}

const BASE_URL = `${BASE_ENDPOINT}/api/v1`;

export default axios.create({
  baseURL: BASE_URL,
  headers: { 'Content-Type': 'application/json', Accept: 'application/json' },
  withCredentials: true
});
```

This configuration creates a base URL that points to the backend server. When the frontend makes API calls, it automatically adds this base URL to the beginning of each request.

The withCredentials: true setting is important because it allows the frontend to send cookies and session information with each request, which the backend needs for authentication.

13.2 Backend

The backend server is configured to accept requests from the frontend and handle them appropriately. The server sets up CORS (Cross-Origin Resource Sharing) to allow the frontend to communicate with it, and it configures various middleware to process incoming requests.

```
// backend server configuration
private securityMiddleware(app: Application): void {
  app.set('trust proxy', 1);
  app.use(
    cookieSession({
      name: 'session',
      keys: [config.SECRET_KEY_ONE!, config.SECRET_KEY_TWO!],
      maxAge: 24 * 7 * 3600000,
      secure: false,
      sameSite: 'lax'
    })
  );
  app.use(hpp());
  app.use(helmet());
  app.use(
    cors({
      origin: config.CLIENT_URL,
      credentials: true,
      optionsSuccessStatus: 200,
      methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS']
    })
  );
}
```

The CORS configuration is crucial because it tells the backend to accept requests from the frontend's domain. Without this, the browser would block the frontend from making requests to the backend due to security policies. The credentials: true setting allows the backend to receive cookies and authentication information from the frontend.

13.3 WebSocket

In addition to HTTP requests, the frontend and backend communicate in real-time using Socket.IO. This allows for instant updates without the frontend having to constantly ask the

backend for new information.

```
// Frontend socket service
class SocketService {
  socket;

  setupSocketConnection() {
    this.socket = io(BASE_ENDPOINT, {
      transports: ['websocket'],
      secure: true
    });
    this.socketConnectionEvents();
  }

  socketConnectionEvents() {
    this.socket.on('connect', () => {
      console.log('connected');
    });

    this.socket.on('disconnect', (reason) => {
      console.log(`Reason: ${reason}`);
      this.socket.connect();
    });

    this.socket.on('connect_error', (error) => {
      console.log(`Error: ${error}`);
      this.socket.connect();
    });
  }
}
```

The frontend creates a Socket.IO connection to the same server that handles the HTTP requests. This connection automatically tries to reconnect if it gets disconnected, ensuring that real-time communication remains active. The frontend listens for various events from the backend, such as new messages, notifications, or updates to posts.

The backend creates a Socket.IO server that can handle multiple frontend connections simultaneously. It uses Redis to manage these connections across multiple server instances if needed.

```
// Backend Socket.IO setup
private async createSocketIO(httpServer: http.Server): Promise<Server> {
  const io: Server = new Server(httpServer, {
    cors: {
      origin: config.CLIENT_URL,
      methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS']
    }
  });
  const pubClient = createClient({ url: config.REDIS_HOST });
  const subClient = pubClient.duplicate();
  await Promise.all([pubClient.connect(), subClient.connect()]);
  io.adapter(createAdapter(pubClient, subClient));
  return io;
}

private socketIOConnections(io: Server): void {
  const postSocketHandler: SocketIOPostHandler = new SocketIOPostHandler(io);
  const followerSocketHandler: SocketIOFollowerHandler = new SocketIOFollowerHandler(io);
  const userSocketHandler: SocketIOUserHandler = new SocketIOUserHandler(io);
  const chatSocketHandler: SocketIOChatHandler = new SocketIOChatHandler(io);
  const notificationSocketHandler: SocketIONotificationHandler = new SocketIONotificationHandler();
  const imageSocketHandler: SocketIOImageHandler = new SocketIOImageHandler();

  postSocketHandler.listen();
  followerSocketHandler.listen();
  userSocketHandler.listen();
  chatSocketHandler.listen();
  notificationSocketHandler.listen(io);
  imageSocketHandler.listen(io);
}
```

The backend sets up different socket handlers for different types of real-time events. For example, there's a handler for post-related events, another for chat messages, and another for notifications. Each handler listens for specific events and can send updates to all connected frontend clients.

14 The Connection Between Frontend and Backend

When a user does something in the app, like clicking a button, typing, or picking an option, the frontend reacts right away. This action starts the communication between the frontend and backend.

The frontend takes the user's input and gets it ready to send to the backend. This means the data is put into the right format, basic checks are done to make sure it's valid, and the request is packed with the needed details, like login tokens and extra information

Once the frontend has prepared the request, it sends the data over the internet to the backend server. This is done using HTTP requests, which are the standard way web apps talk to each

other online. HTTP is used because it's universal, lightweight, and works across all browsers and servers, making it reliable for sending and receiving data on the web. The request passes through different network systems, like routers and switches, before it reaches the backend.

When the backend gets the request, it quickly sends it to the right place. The routing system works like a traffic controller, checking the request's URL, method, and headers to decide which part of the application should handle it.

Before the backend processes a request, it first runs a security check to make sure the user is really who they say they are and has permission to do what they're asking. This is called authentication, and it usually involves checking the JWT token that came with the request.

The backend takes the token from the request headers, decodes it with a secret key, and checks if it's valid and not expired. If everything is fine, it pulls the user's details from the token and attaches them to the request so the rest of the system knows the user's identity and permissions.

The code pulls the JWT token from the Authorization header, verifies it with the server's secret, and decodes the user info. If valid, it attaches the user data to req.currentUser; if not, it throws an authentication error.

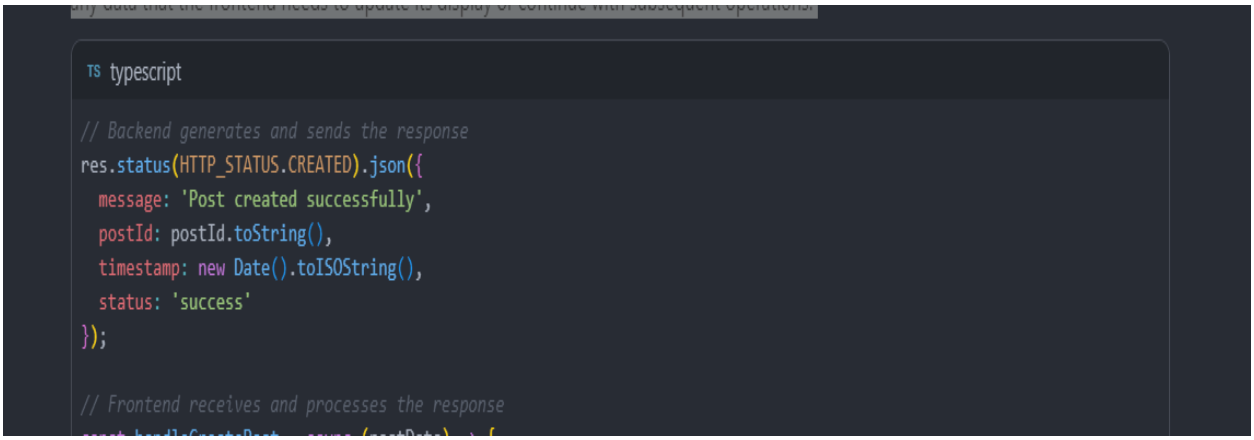
```
// The backend checks if the user is properly authenticated
const authMiddleware = async (req, res, next) => {
  try {
    // Extract the authentication token from the request headers
    const token = req.headers.authorization?.split(' ')[1];

    if (!token) {
      throw new UnauthorizedError('No authentication token provided');
    }

    // Verify the token is valid and decode the user information
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    // Attach the user information to the request for later use
    req.currentUser = decoded;

    // Allow the request to continue to the next step
    next();
  } catch (error) {
    // If authentication fails, return an error response
    next(new UnauthorizedError('Invalid or expired authentication token'));
  }
};
```

Once authentication is confirmed, the backend starts processing the request. The first step is validation, where the backend checks the incoming data to make sure it meets the required rules and follows the correct format.

This step is important for keeping data accurate and preventing errors. The backend makes sure all required fields are included, the data types are correct, the values are within allowed ranges, and that the input doesn't contain anything harmful or invalid.

```
// Backend sends real-time updates to all connected users
socketIO.postObject.emit('add post', {
  _id: postId,
  userId: postData.userId,
  username: postData.username,
  post: postData.post,
  bgColor: postData.bgColor,
  feelings: postData.feelings,
  gifUrl: postData.gifUrl,
  privacy: postData.privacy,
  imgVersion: postData.imgVersion,
  imgId: postData.imgId,
  image: postData.image,
  createdAt: postData.createdAt,
  reactions: {},
  comments: [],
  commentCount: 0,
  reactionCount: 0
});

// Frontend receives and processes real-time updates
socket.on('add post', (newPost) => {
  // Add the new post to the beginning of the posts list
  setPosts(prev => [newPost, ...prev]);

  // Show a notification to the user
  showNotification(`${newPost.username} posted something new`);

  // Update the post count if displayed
  if (setPostCount) {
    setPostCount(prev => prev + 1);
  }

  // Play a subtle sound effect if enabled
```

Once processing is complete, the backend creates a response containing the operation result, any needed data, and status details. It sends this back to the frontend over the same network connection as the original request. The response includes HTTP status codes to show success or failure, error messages if needed, and data for the frontend to update the display



```typescript
// Backend generates and sends the response
res.status(HTTP_STATUS.CREATED).json({
  message: 'Post created successfully',
  postId: postId.toString(),
  timestamp: new Date().toISOString(),
  status: 'success'
});

// Frontend receives and processes the response
```

15 Authentication

When we first started building this social media app, we realized that authentication is the foundation of everything else. Without a secure, reliable authentication system, users can't trust the app with their personal information, and we can't provide personalized experiences. We decided to build the authentication system with multiple layers of security and user-friendly features. This includes traditional email/password authentication, JWT tokens for session management, password reset functionality, and integration with the user profile system. It's like building a secure house with multiple locks, alarms, and a welcoming front door.

16 ✍ Signing Up a New User

When a user wants to create an account, they fill out a registration form with their basic information. The frontend validates the input in real-time and provides immediate feedback to ensure a smooth experience.

When the frontend sends a registration request to the backend, the system runs it through a detailed pipeline of validation, security checks, and data preparation. The backend acts as a gatekeeper, making sure only legitimate and properly formatted requests move forward.

I use a custom signupSchema powered by **Joi** to make sure the username, email, password, avatar color, and avatar image are all correctly formatted and not empty. This helps catch bad or incomplete data early before even touching the database.

@joiValidation(signupSchema)

public async create(req: Request, res: Response): Promise<void> { ... }

After the inputs are validated, I check if the username or email already exists by calling `authService.getUserByUsernameOrEmail(username, email).`
If a user already has that email or username, I immediately throw an error to stop duplicate accounts:

```
const checkIfUserExist: IAuthDocument = await authService.getUserByUsernameOrEmail(username, email);
if (checkIfUserExist) {
  throw new BadRequestError('Invalid credentials');
}
```

If everything looks good, I create two unique **MongoDB ObjectIds**:

- One for the user's authentication record (authObjectId),
- And another for their main user profile (userObjectId).

I also generate a random 12-digit **uId** (User ID) using a small helper function:

```
  if (checkIfUserExist) {
    throw new BadRequestError('Invalid credentials');
  }

  const authObjectId: ObjectId = new ObjectId();
  const userObjectId: ObjectId = new ObjectId();
  const uId = `${Helpers.generateRandomIntegers(12)}`;
```

Next, I build two key objects:

- authData — stores login information like username, email, hashed password, avatar color.
- userDataForCache — stores full profile info like social links, posts count, followers, and more.

```
const authData: IAuthDocument = SignUp.prototype.signupData({
  _id: authObjectId,
  uId,
  username,
  email,
  password,
  avatarColor
});
```

Before saving, I handle the avatar image upload.

Using **Cloudinary**, I upload the avatarImage the user provided, and save the returned image URL for their profile:

```
});
const result: UploadApiResponse = (await uploads(avatarImage, `${userObjectId}`, true, true)) as UploadApiResponse;
if (!result?.public_id) {
  throw new BadRequestError('File upload: Error occurred. Try again.');
}
```

Once the image is successfully uploaded, I attach the profile picture URL to the

Now the user data is ready to be saved.

First, I save it to **Redis cache** for super quick access during login or user lookups. This provides instant user data access without waiting for database operations to complete.

```
await userCache.saveUserToCache(`${userObjectId}`, uId, userDataForCache);
```

Then, I add two background jobs to a **queue system**:

- One job to save the authentication data `(authQueue.addAuthUserJob),`

- One job to save the user profile data (`userQueue.addUserJob`).

This queue-based saving approach makes the sign-up flow really fast for the user without waiting for database writes:

```
// Add to database
authQueue.addAuthUserJob('addAuthUserToDB', { value: authData });
userQueue.addUserJob('addUserToDB', { value: userDataForCache });
```

Finally, after all that, I generate a **JWT token**. Once the password reaches the backend, it goes through extra security processing. The system uses bcrypt, a strong password-hashing algorithm, to securely hash the password before saving it in the database. Bcrypt is designed to be slow and resource-intensive, which makes it resistant to brute-force attacks and rainbow table lookups.

```typescript
const hashedPassword = await bcrypt.hash(req.body.password, 12);
```

```
  const userJwt: string = SignUp.prototype.signToken(authData, userObjectId);
  req.session = { jwt: userJwt };
  res.status(HTTP_STATUS.CREATED).json({ message: 'User created successfully', user: userDataForCache, token: userJwt });
}
```

This token is returned to the frontend along with the new user's data

17 Signing In (Login)

After users create an account, they need a simple and secure way to log in.
When a user signs in, my backend first checks if their username exists, verifies if the password matches, and then generates a secure JWT token to keep them logged in. The process is built to be safe, fast, and make sure only real users with the correct details can access the app.
If everything checks out, the server sends back the user's full profile and a token they'll use to stay authenticated across the app.

Once the user submits the login form, here's exactly what happens step-by-step:

First, their input gets validated with joiValidation(loginSchema) to make sure fields like username and password are properly formatted.

```
@joiValidation(loginSchema)
```

```
const existingUser: IAuthDocument = await
authService.getAuthUserByUsername(username);
```

If no user is found, the server immediately throws an error like "Invalid credentials."

If a user is found, the next step is to check if the password matches.
This uses the comparePassword method inside the Auth schema (which securely compares hashed passwords).

```
const passwordsMatch: boolean = await existingUser.comparePassword(password);
if (!passwordsMatch) {
  throw new BadRequestError('Invalid credentials');
}
```

Again, if the password doesn't match, the server throws an "Invalid credentials" error.

If the username and password are both correct, now it's time to generate a **JWT token**.

This token contains important information like the user's ID, username, email, and avatar color.

```
const userJwt: string = JWT.sign(
  {
    userId: user._id,
    uId: existingUser.uId,
    email: existingUser.email,
    username: existingUser.username,
    avatarColor: existingUser.avatarColor
  },
  config.JWT_TOKEN!
);
```

The server saves this token into the user's session:

```
req.session = { jwt: userJwt };
```

Finally, the server sends a success response back to the frontend, including the user's full details and the newly created JWT token:

When a user clicks logout, the system immediately destroys their session on the server and clears all their login information. The server sends back a success message with empty user data and an empty token, confirming the logout was successful.The frontend removes all stored user information from local storage and clears the user's state from the application.The user is then redirected to the login page, completely logged out and ready to sign in again if needed.

TS typescript

```
export class SignOut {
  public async update(req: Request, res: Response): Promise<void> {
    req.session = null;
    res.status(HTTP_STATUS.OK).json({ message: 'Logout successful', user: {}, token: '' });
  }
}
```

18 🛠 Forgot Password / Reset Password

Sometimes users forget their password, so I needed a way to let them reset it safely.
The flow works like this: the user asks for a reset, the server checks if their email exists, generates a secure token, sends them a reset link through email, and when they click it, they can set a new password.
This whole process is built with security in mind — the token is temporary, and password changes are properly validated before updating anything in the database.

18.1 Step-by-Step Breakdown:

- **1. User Requests Password Reset**

When a user clicks "Forgot Password" and enters their email, the server first validates it.

```
@joiValidation(emailSchema)
```

Then it checks if that email actually exists in the database:

```
const existingUser: IAuthDocu
ment = await authService.getAuthUserByEmail(email);
```

```
If no matching email is found, the server immediately throws an error like
"Invalid credentials."
```

- **2. Generate a Secure Reset Token**

If the email is valid, the server creates a random, secure token using `crypto`.

```
const randomBytes: Buffer = await Promise.resolve(crypto.randomBytes(20));
const randomCharacters: string = randomBytes.toString('hex');
```

This token acts like a *temporary password key* that only works for a short time (it expires after an hour). Then the server updates the user's record to save the token and its expiration time:

```
await authService.updatePasswordToken(`${existingUser._id!}`, randomCharacters,
Date.now() * 60 * 60 * 1000);
```

- **3. Send Password Reset Email**

Now the important part: sending the user an email with the reset link. First, a reset link is created using the token:

```
const resetLink = `${config.CLIENT_URL}/reset-password?token=${randomCharacters}`;
```

Then the server builds an email template using EJS

```
const template: string =
forgotPasswordTemplate.passwordResetTemplate(existingUser.username!, resetLink);
```

Finally, it sends the email by adding it to the `emailQueue` so the email system (SendGrid in production or Nodemailer locally) handle it:

```
emailQueue.addEmailJob('forgotPasswordEmail', { template, receiverEmail: email,
subject: 'Reset your passwor
```

📧 Sending Emails

Handling emails was an important part of the password reset system. I didn't just want users to reset their password — I wanted them to **feel guided** through it, with clear emails at every step. To make this happen, I set up two different emails:

- A **Password Reset Link** email when they request a reset
- A **Confirmation Email** after they successfully change their password

---

19 🛠️ How I Built the Email System:

- **Dynamic Templates with EJS**

Our email system uses **EJS (Embedded JavaScript)** templates to create beautiful, professional emails. EJS is a simple templating language that lets you write HTML with embedded JavaScript

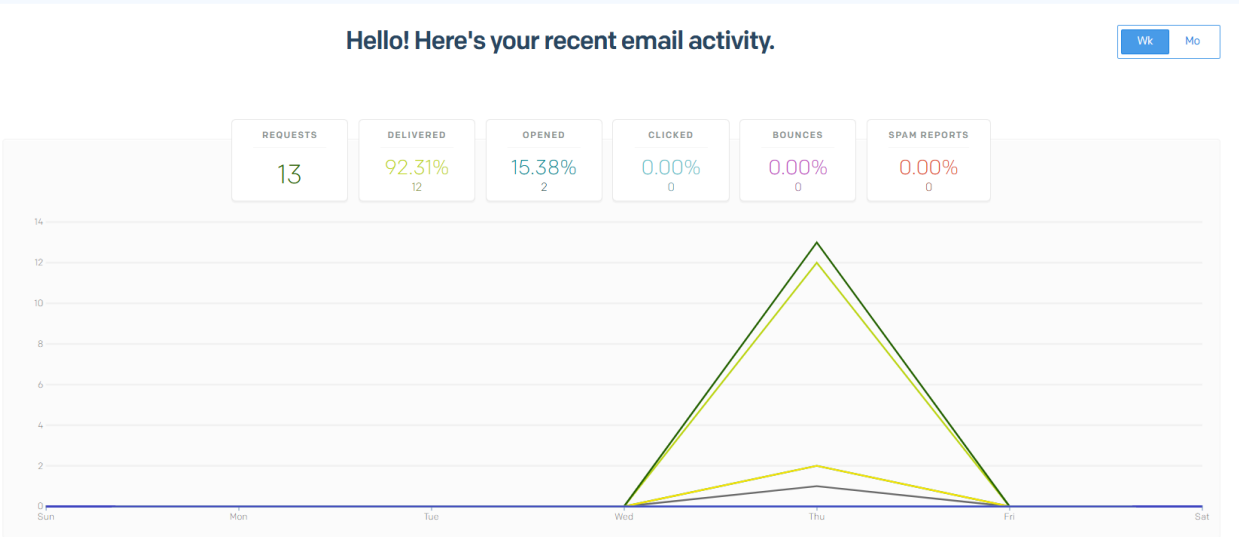
code. This means you can create dynamic emails that include user names, personalized links, and other custom information.

EJS templates are HTML files with special tags for example "**<%= resetLink %>**" this would get replaced with actual data by the worker when the email is created. The system reads the template file, fills in the dynamic parts with user data, and creates a complete HTML email that looks professional and personalized.

```typescript
// Creating a password reset email template with EJS
public passwordResetTemplate(username: string, resetLink: string): string {
  return ejs.render(fs.readFileSync(__dirname + '/forgot-password-template.ejs', 'utf8'), {
    username,
    resetLink,
    image_url: 'https://w7.pngwing.com/pngs/120/102/png-transparent-padlock-logo-computer-icons-padlock-technic-logo-pas
  });
}
```

- **How Emails Are Queued and Sent**

The system uses BullMQ (which works with Redis) to manage queues. We chose BullMQ because it can handle jobs reliably, retry them automatically if they fail, set priorities, and let us monitor jobs in real time.



This means that if the email service goes down for a short time, the system won't lose any emails, they'll just stay in the queue and get sent out automatically once the service is available again.

```
// Email queue configuration
class EmailQueue extends BaseQueue {
  constructor() {
    super('emails');
    this.processJob('forgotPasswordEmail', 5, emailWorker.addNotificationEmail);
    this.processJob('commentsEmail', 5, emailWorker.addNotificationEmail);
    this.processJob('followersEmail', 5, emailWorker.addNotificationEmail);
    this.processJob('reactionsEmail', 5, emailWorker.addNotificationEmail);
    this.processJob('directMessageEmail', 5, emailWorker.addNotificationEmail);
    this.processJob('changePassword', 5, emailWorker.addNotificationEmail);
  }

  public addEmailJob(name: string, data: IEmailJob): void {
    this.addJob(name, data);
  }
}
```

---

20 Global Helpers

As the backend of this social media platform started to grow, I realized early on that there were a lot of repeating patterns happening across different parts of the project — things like checking if a user is authenticated, formatting strings, validating inputs, or throwing custom error messages. These weren't feature-specific.

Rather than copy-pasting the same logic in different controllers or services, I decided to create a global helpers system inside the shared/globals/helpers/ folder. This part of the backend is all

about **cleaning up repetitive tasks**, **enforcing consistency**, and **making the codebase more readable and maintainable over time**.

There are three main components here: **authentication middleware**, **custom error handling**, and **general-purpose utility functions**.

- **Authentication Middleware – Protecting Routes Securely**

The auth-middleware.ts file contains the code that checks if users are logged in before they can access certain parts of the application. It has two main functions that work together to keep the app secure called AuthMiddleware that exposes two main methods: verifyUser() and checkAuthentication().

The verifyUser() method checks if the request session contains a valid JWT. **Remember, a new token is generated every time someone signs in using the user data + secret key.** If it does, the token is verified and decoded, and the user's payload is attached to the request for further use.

```
export class AuthMiddleware {
  public verifyUser(req: Request, _res: Response, next: NextFunction): void {
    if (!req.session?.jwt) {
      throw new NotAuthorizedError('Token is not available. Please login again.');
    }

    try {
      const payload: AuthPayload = JWT.verify(req.session?.jwt, config.JWT_TOKEN!) as AuthPayload;
      req.currentUser = payload;
    } catch (error) {
      throw new NotAuthorizedError('Token is invalid. Please login again.');
    }
    next();
  }
```

This method ensures that **every request has to pass the token check before continuing**. If anything fails, a custom NotAuthorizedError is thrown.

checkAuthentication, It just checks if user information exists in the request. This is useful for routes that need user data but might be called after the initial token check.

- **Custom Error Handling – Making Errors Clear and Consistent**

I didn't want to just rely on generic JavaScript errors like throw new Error('something went wrong'). That kind of error handling gets messy quickly and doesn't give consistent feedback to the client. So instead, I built a structured custom error system in error-handler.ts, centered around a base class called CustomError.

Each error extends this base class and defines its own HTTP status code and message:

```
export class NotAuthorizedError extends CustomError {
  statusCode = HTTP_STATUS.UNAUTHORIZED;
  status = 'error';

  constructor(message: string) {
    super(message);
  }
}
```

This makes it super easy to throw clear, structured errors from anywhere in the app — whether it's a validation failure, missing data, a route not found, or unauthorized access. I also created several other custom errors for specific scenarios:

- BadRequestError
- NotFoundError
- JoiRequestValidationError
- FileTooLargeError
- ServerError

Every single one of these extends CustomError and includes a serializeErrors() method that ensures all errors look the same when sent to the frontend:

```
{
  "message": "Token is not available. Please login again.",
```

```
  "status": "error",
  "statusCode": 401
}
```

This consistency helped massively when it came to debugging or catching client-side errors — the structure never changed, no matter where the error came from.

20.1 Helpers.ts

The helpers.ts file contains useful functions that solve common problems. These functions are used in many parts of the app.

String Formatting Functions: Making Text Look Right

The app needs to format text properly, especially usernames and emails.

```
static firstLetterUppercase(str: string): string {
  const valueString = str.toLowerCase();
  return valueString
    .split(' ')
    .map((value: string) => `${value.charAt(0).toUpperCase()}${value.slice(1).toLowerCase()}`)
    .join(' ');
}
```

Making Emails Lowercase

```
static lowerCase(str: string): string {
  return str.toLowerCase();
}
```

This function makes emails lowercase so they're consistent.

21 Random Number Generation: Creating Unique IDs

The app needs to create unique numbers for user IDs.

```
static generateRandomIntegers(integerLength: number): number {
  const characters = '0123456789';
  let result = ' ';
  const charactersLength = characters.length;
  for (let i = 0; i < integerLength; i++) {
    result += characters.charAt(Math.floor(Math.random() * charactersLength));
  }
  return parseInt(result, 10);
}
```

This function creates random numbers. It's used to make unique user IDs when people sign up.

22 Safe Data Processing: Handling Different Data Types

The app needs to handle different types of data safely. This function tries to parse JSON data. If it fails, it returns the original string instead of crashing.

Checking if Data is an Image

```
static parseJson(prop: string): any {
  try {
    JSON.parse(prop);
  } catch (error) {
    return prop;
  }
  return JSON.parse(prop);
}
```

This function checks if a string is a valid image data URL. It's used when people upload images.

```
TS typescript
static isDataURL(value: string): boolean {
  const dataUrlRegex = /^\s*data:([a-z]+\/[a-z0-9-+.]+(;[a-z-]+=[a-z0-9-]+)?)?(;base64)?,([a-z0-9!$&',()*+;=\-._~:@\/?%
  return dataUrlRegex.test(value);
}
```

23 Array and String Tools: Working with Lists and Text

The app needs tools to work with lists and text. **Shuffling Lists**

```
static shuffle(list: string[]): string[] {
  for (let i = list.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));
    [list[i], list[j]] = [list[j], list[i]];
  }
  return list;
}
```

This function mixes up the order of items in a list. **It's used to show random user suggestions.**

Making Text Safe for Search

```typescript
static escapeRegex(text: string): string {
  return text.replace(/[-[\]{}()*+?.,\\^$|#\s]/g, '\\$&');
}
```

This function makes text safe for search. It prevents special characters from breaking search functions.

- **Why This Matters**

All of this logic — the middleware, the error handling, the utility methods — could've just lived inside controller files or been repeated across services. But by abstracting it all into the helpers/ folder, I ended up with a backend that's **cleaner, more consistent, and easier to maintain**.

If I need to change how errors are formatted, I update one file. If I decide to add new validation to the auth middleware, I do it in one place. And if I reuse a string transformation, I know it behaves the same everywhere.

It also makes onboarding easier for anyone else reading or contributing to the project. They'll know exactly where to look for shared logic, and they won't have to guess whether each controller has its own version of a random generator or error class.

In the long run, it's the kind of small design choice that ends up making a big difference — and it's one of the things I'm proudest of in how I structured this backend.

24 Workers

At a certain point in building this app, I started noticing something: the more features I added, the slower things started to feel. Not in a way that completely broke the app, but just enough to notice. When users posted something, or when an image was uploaded or a notification was triggered, there was a delay. The controllers were doing too much — validating input, saving data, updating multiple collections, maybe even sending emails — all in the same request. It worked, but it wasn't efficient. That's when I realized I needed a better way to handle things.

I wanted users to interact with the app in real time and not be stuck waiting for everything else to finish in the background. If someone reacts to a post or follows another user, that should be instant. The app should respond right away, and all the backend work — whether that's saving to the database or sending a notification — should happen behind the scenes. That's exactly what workers helped me solve. Instead of blocking the main thread, I could push these tasks to a queue and let a background worker handle them asynchronously.

This also solved another problem I was starting to run into — stability. Sometimes, if a task failed (like saving a comment or sending an email), it would crash the entire request. That's not ideal, especially for a growing app. With workers, I can log failures, retry them, or inspect them through Bull Board without affecting the user at all. I also get cleaner logs and a better understanding of where things fail, which made debugging much easier.

So overall, moving to a background worker system wasn't just a performance upgrade — it was a structural one. It made the app feel faster for users, more reliable under pressure, and way easier to scale over time. Controllers are now focused purely on handling the request and queuing up what needs to happen, while the actual "work" — whether that's saving a post, sending an email, or updating user data — is done by dedicated workers running behind the scenes.

25 MongoDB

25.1 ◇ How MongoDB Fits Into the Backend

I used Mongoose to connect my Node.js backend to MongoDB. Mongoose is an Object Document Mapper (ODM) that helps structure how data is stored, giving MongoDB a schema-like feel without removing its flexibility. For every major feature — like posts, comments, users,

messages — I created a Mongoose model that defines how that data should be structured and validated.

Before anything gets saved to MongoDB, the data is first validated using Joi. This makes sure bad or broken data never even reaches the database. After validation, the request is handled by a service file that uses Mongoose to run the actual database logic — things like saving, updating, or deleting a document.

25.2 ◇ MongoDB Connection Setup

The MongoDB connection is handled during server startup using a dedicated server class I built called ChattyServer. All environment variables (like the database URI) are stored in a .env file and loaded through a config utility. This keeps credentials secure and makes it easy to switch between local development and production.

When the app starts, the database is connected like this:

```
const httpServer: http.Server = new http.Server(app);

await mongoose.connect(config.MONGO_URI);
```

From that point, all models are active and mapped to their collections in MongoDB.

25.3 ◇ The Layered Project Blueprint

To keep everything scalable and maintainable, each feature in my project is built using the same consistent structure:

26 ✅ Model

This is the blueprint for how data is saved in MongoDB. For example, the UserModel might include fields like username, email, and followersCount. Mongoose handles all the behind-the-scenes work to store and retrieve that data.

27 ✅ Service

Services are where the business logic lives. Instead of writing database queries in the route or controller, I write them here. Services handle tasks like creating a user, updating a post, or fetching comments.

28 ✅ Validation Schema (Joi)

Before any data touches MongoDB, it's validated using Joi. This ensures things like emails, passwords, and required fields are formatted correctly. If the data doesn't pass, the request is rejected before it can cause any issues.

29 ✅ Worker (Optional)

For tasks that take longer — like sending emails, uploading images, or saving notifications — I use background workers. These jobs are added to a queue and processed in the background using BullMQ, so the app stays fast and responsive.

29.1 User Schema — Storing Everything About the User

The User schema holds all the personal and profile-related data for each user in the app. It's separate from the authentication stuff like passwords (which lives in the Auth schema) and focuses on what users see in their profiles.

One part I really liked designing here was the social section. It stores all the user's social media links in one object:

```
social: {
  facebook: { type: String, default: '' },
  instagram: { type: String, default: '' },
  twitter: { type: String, default: '' },
```

```
  youtube: { type: String, default: '' }
}
```

I grouped all these fields together instead of scattering them as top-level fields, just to keep things clean and easy to manage. If someone doesn't fill in one of their links, it just stays as an empty string. That way, the frontend won't break trying to display a missing value.

Besides that, the schema also includes things like:

- Profile picture
- Posts count
- Follower/following counts
- Background image
- Bio fields (quote, work, school, location)
- Notification settings (to control things like whether they get alerts for messages or follows)
- Block lists (who they blocked and who blocked them)

This schema basically builds out the full user profile and gives the app a ton of flexibility for personalization.

29.2 Post Schema — User-Generated Content

Posts are at the heart of this project — they show up in the feed, can have comments, reactions, and media attached. The most important part here is how I linked each post to a specific user:

```
userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
```

This ref: 'User' tells Mongoose that this field connects to a document in the User collection. So when I fetch a post, I can also pull in that user's username, avatar, or profile picture using .populate(). That made the feed feel more complete without needing extra queries.

Other useful fields in this schema include:

- post: the actual text content
- bgColor: lets users customize their post background
- imgVersion / imgId / videoVersion / videoId: tracks media uploads (especially useful for Cloudinary)
- gifUrl: supports embedding a GIF into the post
- privacy: lets users control who can see it
- feelings: stores an emotion or mood (like “excited” or “grateful”)
- reactions: stores the count of each type of emoji reaction (like, love, wow, etc.)
- commentsCount: saves how many comments it has

This schema gave me everything I needed to build a clean social media-style feed.

29.3 💬 Conversation Schema — Private Chats Between Users

The Conversation schema represents a single chat thread between two users. It looks simple, but it’s important:

```
senderId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },

receiverId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
```

Both senderId and receiverId are stored as ObjectId values and point to users. That way, I can easily find all the conversations a user is part of, and even join them to usernames or profile pictures if I need to.

Using ObjectId with ref keeps the database efficient and makes lookups fast. I also added indexes here to speed up search when checking if a conversation already exists.

If I wanted to support group chats later, this schema is the one I'd expand by adding an array of participant IDs.

29.4 📨 Message Schema — Storing Each Message in a Conversation

Messages are linked to conversations. Each message belongs to a specific thread (conversationId), and includes info about who sent it, who received it, and the content.

One field I added that made the messaging experience more fun was gifUrl:

```
gifUrl: { type: String, default: '' }
```

This field stores a link to a GIF (usually from Giphy) that users can send in chat. It's optional — if no GIF is sent, it stays empty. But if a user sends a GIF-only message, it still shows up in the thread just like a regular message.

This gave chat a bit more personality without overcomplicating the message structure.

Other fields in the message schema include:

- body: for text content
- selectedImage: in case the message includes a photo
- isRead: used for showing read receipts
- reaction: lets users react to messages (emoji reactions, like in DMs)
- deleteForMe / deleteForEveryone: controls who can still see the message

This schema covers every message scenario I needed — from basic texts to rich media, and even soft-deletion.

29.5 ◇ Summary

This setup makes the backend much easier to scale and maintain. By separating concerns — validation, business logic, data models, and background tasks — I can quickly make changes to one part of the system without breaking anything else.

MongoDB acts as the solid foundation for the entire app, managing storage, relationships, and performance across every feature.

30 Service Layer: Handling the Logic Between Backend and Database

Before anything could happen on the backend — like signing up users, posting content, chatting, or sending notifications — I needed a solid way to actually talk to the database cleanly.

That's where my **service layer** came in.
These services are basically the workers behind the scenes.
Every time the app needed to **create**, **read**, **update**, or **delete** something in **MongoDB**, it was done through these database services.
Each one is focused on a different feature (like users, posts, or messages), which kept my backend organized, scalable, and easy to maintain later.

The service layer made sure my API routes stayed lightweight — they just called the right service — while all the heavy database logic happened neatly inside these classes.

Here’s a breakdown of the services I built:

30.1 🔓 Auth Service — Managing Authentication Data

The AuthService handles everything around user credentials — creating a user during signup, managing password reset tokens, and fetching users by username or email.

Some key methods:

- `createAuthUser(data)`: Save a new user's credentials.
- `updatePasswordToken(authId, token, expiration)`: Save a password reset token for when users forget their passwords.
- `getUserByUsernameOrEmail(username, email)`: Search by username or email (useful during login).
- `getAuthUserByPasswordToken(token)`: Used during password resets to check if the token is still valid.

Keeping this all inside a service made auth-related stuff clean and safe.

30.2 🚫 Block User Service — Handling Block/Unblock Actions

The BlockUserService is responsible for when a user blocks or unblocks someone.

Main features:

- `blockUser(userId, followerId)`: Adds someone to your block list.
- `unblockUser(userId, followerId)`: Removes someone from the block list.

It uses bulkWrite() to update both users at once, making it super fast and clean.

30.3 💬 Chat Service — Real-Time Messaging

The ChatService controls messaging between users.

It handles:

- Saving new conversations if they don't exist yet.
- Saving each new message, including text, GIFs, or images.
- Marking messages as read or deleted.
- Handling reactions on messages.

Without this service, chat would have gotten messy really fast.

30.4 💬 Comment Service — Managing Post Comments

The CommentService takes care of:

- Adding new comments to posts.
- Updating the comments count on posts.
- Sending real-time notifications and emails when someone comments.

It bundles everything that needs to happen when a user leaves a comment, making it automatic and efficient.

30.5 ➕ Follower Service — Follow and Unfollow Logic

The FollowerService manages:

- Following a new user (and updating counts for both users).
- Unfollowing users.
- Fetching detailed follower/followee lists.

It also handles sending a notification when someone follows you, making the app feel more interactive.

30.6 🖼️ Image Service — Uploading and Managing Images

The ImageService deals with:

- Saving profile and background images.
- Storing image metadata (like Cloudinary IDs).
- Fetching and deleting images when needed.

By splitting images into their own service, everything stayed organized without cluttering user data.

30.7 🔔 Notification Service — Keeping Users Updated

The NotificationService handles:

- Fetching all of a user's notifications (new followers, comments, likes, etc.).
- Marking notifications as read.
- Deleting notifications.

It made it super easy to manage notifications in the backend and frontend.

30.8 📝 Post Service — Handling User Posts

The PostService powers:

- Creating new posts.
- Fetching posts for the feed or a profile.
- Editing or deleting posts.
- Counting total posts.

It handles different types of posts too — text, images, videos, or GIFs — and keeps things flexible.

30.9 💙 Reaction Service — Likes and Reactions

The ReactionService manages:

- Adding new reactions to posts (like a ❤️, 😂, 😲, etc.).
- Removing reactions.
- Fetching reactions for a post or a user.

It also sends out notifications when someone reacts to your post, making user engagement feel instant.

30.10 👩‍💻 User Service — Managing User Profiles and Settings

The UserService takes care of:

- Saving user data after signup.
- Updating profile info (work, school, bio, location).
- Updating social media links.
- Managing notification settings.
- Searching for users.

It also has smart methods like fetching random users for suggestions or showing users you're not already following.

30.11 Final Thoughts on the Service Layer

The database services were honestly one of the best parts of the project structure.

Each service did one job — handled one part of the database — and stayed separate from other

parts of the app.

It made the backend clean, scalable, and much easier to maintain.

If I ever needed to add a feature — like adding a "pin message" option in chat or a "mute notifications" setting — I could just expand the right service without messing up the rest of the codebase.

Building out a strong service layer upfront saved me a ton of time later on and made the app way more professional overall.

31 Real-Time Communication: Socket.IO Setup and Handlers

After setting up the backend and database, I wanted my app to feel alive — like users could get instant updates without refreshing the page. That's where **Socket.IO** came in.

Sockets basically let the server and clients talk to each other live. Whether it's sending a message, showing someone went online, reacting to a post, or updating followers — it happens in real-time without needing to reload anything.

I broke down my socket setup into different handlers based on what feature they covered. Here's how it all worked:

31.1 Basic Socket Setup

At the core, I connected the Express server to **Socket.IO**.

When a user connects to the app, they open a socket connection — kind of like a live tunnel where the server and the frontend can send messages back and forth instantly. After setting up the backend and database, I wanted my app to actually feel alive — like users could get instant updates without refreshing the page. That's where **Socket.IO** came in.

Sockets basically let the server and the client have a constant open connection, so they can send and receive messages **in real time** without having to reload the page or refresh anything.

Every major feature in my app (chat, followers, notifications, user status, images) had its own **Socket Handler** class to keep everything clean and separated.

```
private async createSocketIO(httpServer: http.Server): Promise<Server> {
  const io: Server = new Server(httpServer, {
    cors: {
      origin: config.CLIENT_URL,
      methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS']
    }
  });

  return io;
}
```

To set up real-time communication, I started by creating a Socket.IO server with new Server(httpServer, { cors: {...} }).

I needed to build the socket server **on top of my HTTP server** because Socket.IO works at a lower level than Express — it needs to listen directly to incoming connections, not just API routes.

While setting up the server, I added a **CORS config** to make sure my frontend could connect without the browser blocking it.

The CORS setup allowed only my frontend URL (config.CLIENT_URL) to connect and made sure that common HTTP methods like 'GET', 'POST', 'PUT', 'DELETE', and 'OPTIONS' were accepted.

Without CORS, browsers would throw errors and block the socket connection.

31.2 The Flow of Real-Time Communication

To visualize how sockets were working in my app:

1. **Frontend** opens a socket connection automatically when the app loads.
2. **Server** listens for specific events (like "join room", "unfollow user", "setup user").

3. When an event happens (ex: new chat message, user goes online), the **Server emits** back a custom event.
4. **Frontend listens** for that event and updates the UI instantly.

No page reloads, no waiting for full API calls — **just pure real-time interaction**.

31.3 Chat Socket Handler — Joining Chat Rooms

The SocketIOChatHandler manages live messaging.

When a user starts or opens a chat, their frontend sends a 'join room' event to the server with the usernames of both the sender and receiver.

Inside the handler, I did this:

```
this.io.on('connection', (socket: Socket) => {
  socket.on('join room', (users: ISenderReceiver) => {
    const { senderName, receiverName } = users;
    const senderSocketId: string = connectedUsersMap.get(senderName) as string;
    const receiverSocketId: string = connectedUsersMap.get(receiverName) as string;
    socket.join(senderSocketId);
    socket.join(receiverSocketId);
  });
```

Basically, it joins both users into private "rooms" based on their socket IDs.

This way, when one user sends a message, it instantly goes only to the other user without needing to fetch new messages manually.

It made chat feel smooth and real-time, like any modern messaging app.

31.4 ➕ Follower Socket Handler — Updating Follows

The SocketIOFollowerHandler is for when someone **unfollows** you.

When that happens, the server listens for the 'unfollow user' event:

```
export class SocketIOFollowerHandler {
  private io: Server;

  constructor(io: Server) {
    this.io = io;
    socketIOFollowerObject = io;
  }

  public listen(): void {
    this.io.on('connection', (socket: Socket) => {
      socket.on('unfollow user', (data: IFollowers) => {
        this.io.emit('remove follower', data);
      });
    });
```

And it instantly notifies the client (frontend) to remove the follower from their followers list without refreshing the page.

Simple, but made a huge difference for keeping the app dynamic.

31.5 User Socket Handler — Tracking Online Status

The SocketIOUserHandler was one of the cooler parts — it made tracking **who's online** feel real.

When a user logs in, the frontend sends a 'setup' event, and the server saves their user ID and socket ID into a live map:

```
  connectedUsersMap.set(username, socketId);
}
```

This helped me:

When a user connects to the app, I needed a way to remember them while they were online. That's where the connectedUsersMap comes in — it's basically a live list where I store every user's name and their socket ID (which is like their personal connection ID). The line connectedUsersMap.set(username, socketId); simply adds the user into that list. So now, whenever I need to send real-time updates — like a new chat message or a notification — I know exactly which user to send it to. Without this map, the server wouldn't know who's online or how to reach them instantly.

When a user disconnects, it removes them from the map and updates the frontend again:

```
private removeClientFromMap(socketId: string): void {
  if (Array.from(connectedUsersMap.values()).includes(socketId)) {
    const disconnectedUser: [string, string] = [...connectedUsersMap].find((user: [string, string]) => {
      return user[1] === socketId;
    }) as [string, string];
    connectedUsersMap.delete(disconnectedUser[0]);
    this.removeUser(disconnectedUser[0]);
    this.io.emit('user online', users);
  }
}
```

When a user disconnects from the app — whether they close the browser, lose connection, or log out — I needed a way to clean up their information from the server side. That's exactly what the removeClientFromMap function handles. First, it checks if the disconnected socket ID exists in the connectedUsersMap, which is where I keep track of all online users. If it does, it finds the matching username tied to that socket ID. Once the correct user is found, it deletes them from the map and also removes them from the users array that holds all currently online users. After cleaning up, it emits a fresh 'user online' event to the frontend, so the online users list updates in real-time for everyone still connected. This small system makes sure the app always shows the correct users online, keeping everything dynamic and accurate without needing any page refreshes.

32 Real-Time Chat System Architecture

The chat system is designed so messages are sent right away to people who are online, while also being saved safely for anyone who is offline. This way, it feels instant, but nothing ever gets lost.

The chat system uses Socket.IO with WebSockets to send messages instantly. When someone connects, their username and socket ID are saved in a connected users map. This map makes it easy to see who is online and send messages directly to them without searching through everyone. If the recipient is online, the message goes to them right away. If they're offline, the system saves the message in the database, so they get it later.

32.1 Message Sending Process

The frontend identifies users in several ways during chat operations. When sending a message, the frontend includes the sender's information (from the JWT token) and the recipient's information (from the chat interface). The backend uses this information to route the message correctly and update the appropriate conversation.

```typescript
// Frontend sends message via HTTP request
const sendMessage = async (messageData) => {
  try {
    const response = await axios.post('/api/chat/message', messageData, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      }
    });

    // Also emit through socket for real-time delivery
    socket.emit('send message', messageData);

    return response.data;
  } catch (error) {
    console.error('Error sending message:', error);
  }
};
```

When requesting chat history or conversation lists, the frontend sends the current user's ID (from the JWT token) so the backend can return only the conversations that belong to that user.

32.2 Real-Time Chat System Architecture

Chat System Overview

The chat system uses a layered design that makes sure messages are delivered instantly to users

who are online, while also saving them safely for users who are offline. This gives both quick responsiveness and reliable storage.

Socket.IO for Real-Time Messaging
Real-time messaging is powered by **Socket.IO**, which uses WebSocket connections for instant delivery. The system keeps a **map of connected users** that links each username with their socket ID. This works like a lookup table, letting the backend quickly find and send messages to the right person without checking through everyone.

This map is important because it tells the system exactly who is online and how to reach them. If the recipient is online, the message is delivered immediately. If they're offline, the message is stored in the database so it can be retrieved later.

32.3 Why use Socket.IO

The key difference between a database-only approach and Socket.IO is how messages are delivered: with a database, the client has to keep sending requests asking "any new messages?", which causes delays, high server load, wasted bandwidth, and missing real-time features. With Socket.IO, the connection stays open and messages are pushed instantly to users as soon as they happen.

Using only a database would lead to delayed delivery, missing real-time features, and poor scalability. Socket.IO fixes this by providing instant delivery and live features, while the database stores messages, chat history, and offline data. This hybrid approach gives both speed and reliability.

32.4 Socket.IO configuration

When a user opens the chat application, the frontend establishes a Socket.IO connection with the backend server. This connection is maintained throughout the user's session and allows for instant communication.

```
// Frontend establishes connection
const socket = io(BASE_ENDPOINT, {
  transports: ['websocket'],
  secure: true
});

// Frontend sends user information to server
socket.emit('setup', {
  userId: profile._id,
  username: profile.username
});
```

On the backend, the server listens for these connection events and manages the connected users. When a user connects, their information is stored in a map that tracks all online users. This map is crucial for routing messages to the correct recipients.

```
export class SocketIOUserHandler {
  public listen(): void {
    this.io.on('connection', (socket: Socket) => {
      socket.on('setup', (data: ILogin) => {
        this.addClientToMap(data.userId, socket.id);
        this.addUser(data.userId);
        this.io.emit('user online', users);
      });

      socket.on('disconnect', () => {
        this.removeClientFromMap(socket.id);
      });
    });
  }
}
```

The most important use of Socket.IO in this project is for instant message delivery. When a user sends a message, it's delivered immediately to online recipients through the Socket.IO connection. The system works by checking if the message recipient is online using the connected users map. If they are online, the message is sent directly to their socket connection. If they are offline, the message is stored in the database for later retrieval.

```
export class SocketIOUserHandler {
  public listen(): void {
    this.io.on('connection', (socket: Socket) => {
      socket.on('setup', (data: ILogin) => {
        this.addClientToMap(data.userId, socket.id);
        this.addUser(data.userId);
        this.io.emit('user online', users);
      });

      socket.on('disconnect', () => {
        this.removeClientFromMap(socket.id);
      });
    });
  }
}
```

32.5 Chat Queue

The chat system uses a background processing setup to handle tasks like saving messages, sending notifications, and other operations that don't need to happen instantly. This keeps the main application fast and responsive, while still making sure that all data is stored and processed correctly.

The chat queue takes care of saving messages to the database, updating read statuses, handling deletions, and processing reactions. It improves reliability by making sure messages are still processed even if the system has temporary issues, and it supports scalability by letting multiple workers process tasks at the same time. It also boosts performance by keeping heavy database operations out of the real-time message flow. On top of this, the queue provides monitoring tools so developers can track background tasks and quickly spot any issues, helping keep the system healthy and consistent.

```
class ChatQueue extends BaseQueue {
  constructor() {
    super('chats');
    this.processJob('addChatMessageToDB', 5, chatWorker.addChatMessageToDB);
    this.processJob('markMessageAsDeletedInDB', 5, chatWorker.markMessageAsDeleted);
    this.processJob('markMessagesAsReadInDB', 5, chatWorker.markMessagesAsReadInDB);
    this.processJob('updateMessageReaction', 5, chatWorker.updateMessageReaction);
  }

  public addChatJob(name: string, data: IChatJobData | IMessageData): void {
    this.addJob(name, data);
  }
}
```

32.6 Chat Worker: Processing Background Tasks

The chat worker is the part of the system that actually runs the queued jobs. It handles things like adding new messages into the database, marking messages as read, deleting messages, and updating reactions. I wrote it so that each method takes in a `job`, pulls out the data from `job.data`, runs the right function in `chatService`, then updates the progress and marks the job as done. This way the heavy database work runs in the background, and the real-time chat isn't slowed down.

One thing I had to be careful about was error handling. In my first version, `addChatMessageToDB` called `done(null, job.data)` even if something failed, which made the queue think everything was fine and skipped retries. That could lose messages. I changed it so errors are passed back with `done(error)` which lets BullMQ retry the job. To make retries safe, I also pass a unique `messageId` so the database can ignore duplicates if the same job runs twice. For marking messages as read, the pattern was already correct, since it failed properly on errors. I also added progress updates and better logging so I can track jobs and debug easier. This way the worker keeps the chat system reliable and consistent, even if something breaks in the middle.

```
class ChatWorker {
  async addChatMessageToDB(job: Job, done: DoneCallback): Promise<void> {
    try {
      const { senderId, receiverId, conversationId, message } = job.data;
      await chatService.addChatMessageToDB(senderId, receiverId, conversationId, message);
      job.progress(100);
      done(null, job.data);
    } catch (error) {
      log.error('Chat worker error:', error);
      log.info('Chat job completed with errors but continuing');
      done(null, job.data);
    }
  }

  async markMessagesAsReadInDB(job: Job, done: DoneCallback): Promise<void> {
    try {
      const { senderId, receiverId } = job.data;
      await chatService.markMessagesAsRead(senderId, receiverId);
      job.progress(100);
      done(null, job.data);
    } catch (error) {
      log.error(error);
      done(error as Error);
    }
  }
}
```

32.7 Chat React

Users can react to messages with emojis, providing a more interactive and expressive communication experience. This feature allows users to quickly respond to messages without typing a full response, making conversations more dynamic and engaging. The reaction system is implemented through the background processing queue, ensuring that reactions are stored reliably while still being delivered in real time to other users in the conversation.

```
export class Add {
  public async reaction(req: Request, res: Response): Promise<void> {
    const { messageId, type } = req.body;

    chatQueue.addChatJob('updateMessageReaction', {
      messageId,
      type,
      senderName: req.currentUser!.username,
      senderProfilePicture: req.currentUser!.profilePicture
    });

    res.status(HTTP_STATUS.OK).json({ message: 'Reaction added' });
  }
}
```

In the code, the `reaction` method receives the reaction details from the request body, such as the `messageId` and the emoji `type`. It then pushes a job called `updateMessageReaction` into the `chatQueue`, which includes the reaction data along with the sender's username and profile picture. This job is later picked up by a worker to update the database and broadcast the reaction in real time. Finally, the API responds with a confirmation message, letting the client know that the reaction has been added.

32.8 Chat delete

Users can delete messages for themselves or for everyone in the conversation, giving them control over their communication and privacy. This feature is important for maintaining privacy and letting users correct mistakes or remove inappropriate content. The deletion system supports two modes: deleting a message only for the sender (deleteForMe) and deleting a message for everyone in the conversation (deleteForEveryone). This flexibility allows users to choose the right level of message removal based on their needs.

In the code, the markMessageAsDeleted method reads the messageId and the deletion type from the request parameters. It then pushes a job called markMessageAsDeleted into the chatQueue, passing along the details. A background worker later processes this job, updates the database to mark the message as deleted, and ensures that the change is reflected for the correct users. Once the job is queued, the API responds with a confirmation message so the client knows the deletion request was accepted.

```
export class Delete {
  public async markMessageAsDeleted(req: Request, res: Response): Promise<void> {
    const { messageId, type } = req.params;

    chatQueue.addChatJob('markMessageAsDeleted', {
      messageId,
      type
    });

    res.status(HTTP_STATUS.OK).json({ message: 'Message deleted' });
  }
}
```

33 Followers

The followers functionality in this application manages how users connect with each other on the social platform. It handles actions like following, unfollowing, blocking users, and keeping track

of follower relationships. The system is designed to work in real time, so users see updates and changes immediately.

33.1 Followers' workings

The system uses a database to store all the follow relationships between users. Each relationship is stored as a document with two important pieces of information: who is following and who is being followed.

```
// This is how the system stores follow relationships
const followerSchema: Schema = new Schema({
  followerId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', index: true },
  followeeId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', index: true },
  createdAt: { type: Date, default: Date.now() }
});
```

The followerId is the person who is doing the following, and the followeeId is the person being followed. This simple structure lets the system track all relationships between users.

33.2 Socket.iO

When user clicks follow or unfollow, the system immediately shows the change on the screen. This happens because the system uses Socket.IO to send instant messages between the server and all connected users.

```
// When a user unfollows someone, this code runs
socket.on('unfollow user', (data: IFollowers) => {
  this.io.emit('remove follower', data);
});
```

The system listens for follow and unfollow actions and immediately tells all connected users about the change:

33.3 Follow/Unfollow Processing Flow

When a user follows or unfollows someone, the system does several things in order to make sure the change happens quickly and is saved properly.

34 Step 1: Update the Cache First

The system immediately updates the cache to show the change right away:

```
// Update follower counts in cache immediately
const followersCount: Promise<void> = followerCache.updateFollowersCountInCache(
  `${followerId}`, 'followersCount', 1
);
const followeeCount: Promise<void> = followerCache.updateFollowersCountInCache(
  `${req.currentUser!.userId}`, 'followingCount', 1
);
```

This makes the change appear instantly on the user's screen.

Step 2: Send Real-Time Update

The system then sends a message to all connected users about the change:

```
// Send real-time update to all users
const addFolloweeData: IFollowerData = Add.prototype.userData(response[0]);
socketIOFollowerObject.emit('add follower', addFolloweeData);
```

This ensures everyone sees the change immediately.

35 Step 3: Save to Database in Background

The system adds the change to a background queue so it gets saved to the database without slowing down the user experience

```
// Add the follow action to background processing
followerQueue.addFollowerJob('addFollowerToDB', {
  keyOne: `${req.currentUser!.userId}`,
  keyTwo: `${followerId}`,
  username: req.currentUser!.username,
  followerDocumentId: followerObjectId
});
```

35.1 Queue and Worker System

The followers system uses Queue to save data without making users wait. This means users see changes immediately while the system saves everything properly in the background. The queue system handles saving follow relationships to the database. It works like a waiting line where tasks get processed one by one:

The worker handles the actual work of saving follower data to the database. The database stores all relationships permanently so they don't get lost if the system restarts:

When a user follows someone, the job is added to the queue and the worker processes it in the background. This ensures the system stays fast and responsive while still keeping all follower information accurate.

```typescript
async addFollowerToDB(job: Job, done: DoneCallback): Promise<void> {
  try {
    const { keyOne, keyTwo, username, followerDocumentId } = job.data;
    await followerService.addFollowerToDB(keyOne, keyTwo, username, followerDocumentId);
    job.progress(100);
    done(null, job.data);
  } catch (error) {
    log.error(error);
    done(error as Error);
  }
}
```

In the code, the addFollowerToDB function runs when someone follows another user. The queue passes a **job** into this function. That job contains the details it needs, like:

- keyOne and keyTwo → the two users involved in the follow action
- username → the name of the user who is following
- followerDocumentId → the ID of the follower record in the database

The worker then calls followerService.addFollowerToDB, which saves this new follow relationship into the database. After it finishes, the worker marks the job as **100% complete** so the system knows it worked.

If something goes wrong, the error is logged and the job is marked as **failed**. That way the queue can **retry** it later instead of losing the follow action. On top of saving the follow relationship, this process can also update the follower/following counts and send a notification to the user who just got a new follower.

35.2 Cache

The cache system in this application is like having a super-fast memory that stores information the app uses frequently. Instead of going to the main database every time someone wants to see a user's profile or posts, the app keeps a copy of this information in a special fast storage area called Redis. This makes everything much faster because reading from memory is thousands of times quicker than reading from a hard drive.

When a user first logs into the application, the system takes their information from the database and saves it in the cache. This includes things like their username, email, profile picture, follower count, and other details. The cache organizes this information in a way that makes it easy to find quickly. For example, user data is stored using the user's ID as a key, so when someone wants to see a profile, the app can instantly look up all the information using that ID.

```
// Save user data to cache for fast access
public async saveUserToCache(key: string, userUId: string, createdUser: IUserDocument): Promise<void> {
  const dataToSave = {
    '_id': `${_id}`,
    'username': `${username}`,
    'email': `${email}`,
    'followersCount': `${followersCount}`,
    'followingCount': `${followingCount}`,
    'profilePicture': `${profilePicture}`
  };

  // Store user in organized hash structure
  for (const [itemKey, itemValue] of Object.entries(dataToSave)) {
    await this.client.HSET(`users:${key}`, `${itemKey}`, `${itemValue}`);
  }
}
```

The cache system uses different storage methods depending on what type of data it's storing. User profiles are stored as "hashes" which are like organized lists where each piece of information has a label. Posts are stored in "sorted sets" which keep them in chronological order so the newest posts appear first. Comments are stored as "lists" which makes it easy to add new comments to the beginning and retrieve them in the correct order. Follower relationships are also stored as lists, making it simple to add or remove followers quickly.

When someone performs an action like following another user, the system immediately updates the cache to show the change. This happens instantly because the cache is stored in the computer's memory. The follower count increases immediately, and the relationship is recorded. Later, in the background, the system also saves this change to the main database to make sure it's permanently stored. This approach gives users instant feedback while ensuring data is safely stored.

```
// Get user data from cache instantly
public async getUserFromCache(userId: string): Promise<IUserDocument | null> {
  try {
    if (!this.client.isOpen) {
      await this.client.connect();
    }

    // Get all user data from cache in one operation
    const response: IUserDocument = (await this.client.HGETALL(`users:${userId}`)) as unknown as IUserDocument;

    // Convert stored strings back to proper data types
    response.followersCount = Helpers.parseJson(`${response.followersCount}`);
    response.followingCount = Helpers.parseJson(`${response.followingCount}`);

    return response;
  } catch (error) {
    log.error(error);
    throw new ServerError('Server error. Try again.');
  }
}
```

36 Posts Functionality: Complete System Architecture and Implementation

36.1 Post functionality

I designed the posts feature as a complete social media content system that covers everything from creating posts to handling real-time interactions and privacy. It supports post creation and management, live reactions and comments, privacy filtering, background processing, and reliable data storage, so users always get a smooth and consistent experience.

I built the system that makes sure posts are created quickly, stored safely, and displayed according to each user's privacy settings. It supports different content types like text, images, videos, and GIFs, while still keeping performance high and the data consistent.

When a user creates a post, the system runs it through a pipeline I designed to handle multiple media formats. First, the content is validated to make sure it's correct and safe. After that, it's saved into both the cache and the database — the cache makes the post instantly available in real time, while the database ensures it's stored permanently for future access.

```
// Post schema structure
const postSchema: Schema = new Schema({
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', index: true },
  username: { type: String },
  email: { type: String },
  avatarColor: { type: String },
  profilePicture: { type: String },
  post: { type: String, default: '' },
  bgColor: { type: String, default: '' },
  imgVersion: { type: String, default: '' },
  imgId: { type: String, default: '' },
  videoVersion: { type: String, default: '' },
  videoId: { type: String, default: '' },
  feelings: { type: String, default: '' },
  gifUrl: { type: String, default: '' },
  privacy: { type: String, default: '' },
  commentsCount: { type: Number, default: 0 },
  reactions: {
    like: { type: Number, default: 0 },
    love: { type: Number, default: 0 },
    happy: { type: Number, default: 0 },
    wow: { type: Number, default: 0 },
    sad: { type: Number, default: 0 },
    angry: { type: Number, default: 0 }
  },
  createdAt: { type: Date, default: Date.now }
```

This structure allows the system to handle various types of content while maintaining engagement metrics and privacy controls.

36.2 How it displays posts

The post display system uses a **chronological approach** that shows all posts from newest to oldest, with filtering based on privacy settings and user relationships. When a user visits their feed, the program fetches **all posts** from the database, not just posts from people they follow.

```
// Backend: Get all posts chronologically
public async getPosts(query: IGetPostsQuery, skip = 0, limit = 0, sort: Record<string, 1 | -1>): Promise<IPostDocument[]
  let postQuery = {};
  if (query?.imgId && query?.gifUrl) {
    postQuery = { $or: [{ imgId: { $ne: '' } }, { gifUrl: { $ne: '' } }] };
  } else if (query?.videoId) {
    postQuery = { $or: [{ videoId: { $ne: '' } }] };
  } else {
    postQuery = query; // Empty query = get all posts
  }

  // Get posts sorted by creation date (newest first)
  const posts: IPostDocument[] = await PostModel.aggregate([
    { $match: postQuery },
    { $sort: sort }, // { createdAt: -1 } = newest first
    { $skip: skip },
    { $limit: limit }
  ]);
  return posts;
}
```

This approach ensures all users see the same chronological feed, creating a unified community experience.

36.3 Filtering

The system filters posts based on privacy settings and user relationships. The checkPrivacy method decides whether a post can be seen by the current user. It checks three cases: if the post is marked Private and belongs to the user, if it's marked Public (so anyone can see it), or if it's for Followers and the current user is following the post owner. If any of these checks are true, the method returns true, meaning the post is visible. Otherwise, it stays hidden.

```
// Check privacy settings to determine visibility
static checkPrivacy(post, profile, following) {
  const isPrivate = post?.privacy === 'Private' && post?.userId === profile?._id;
  const isPublic = post?.privacy === 'Public'; // Anyone can see public posts
  const isFollower = post?.privacy === 'Followers' && Utils.checkIfUserIsFollowed(following, post?.userId, profile?._id)
  return isPrivate || isPublic || isFollower;
}
```

36.4 Comments and Reactions

The system stores reaction counts for each type:

```
// Reaction structure in posts
reactions: {
  like: { type: Number, default: 0 },
  love: { type: Number, default: 0 },
  happy: { type: Number, default: 0 },
  wow: { type: Number, default: 0 },
  sad: { type: Number, default: 0 },
  angry: { type: Number, default: 0 }
}
```

The reaction method handles when a user reacts to a post. It first builds a reactionObject with all the details, including the post ID, type of reaction, username, profile picture, and a unique ID. The reaction is saved straight into the cache so the post updates instantly for everyone. At the same time, I send the reaction data to a background queue, which later saves it into the database without blocking the user. This way, reactions feel real-time and fast, while still being stored permanently in the background.

```
// Add reaction to post
public async reaction(req: Request, res: Response): Promise<void> {
  const { userTo, postId, type, previousReaction, postReactions, profilePicture } = req.body;

  // Create reaction object
  const reactionObject: IReactionDocument = {
    _id: new ObjectId(),
    postId,
    type,
    avataColor: req.currentUser!.avatarColor,
    username: req.currentUser!.username,
    profilePicture
  } as IReactionDocument;

  // Update cache immediately
  await reactionCache.savePostReactionToCache(postId, reactionObject, postReactions, type, previousReaction);

  // Save to database in background
  const databaseReactionData: IReactionJob = {
    postId,
    userTo,
    userFrom: req.currentUser!.userId,
    username: req.currentUser!.username,
    type,
    previousReaction,
    reactionObject
  };
  reactionQueue.addReactionJob('addReactionToDB', databaseReactionData);
  res.status(HTTP_STATUS.OK).json({ message: 'Reaction added successfully' });
}
```

The system uses Socket.IO to send reaction updates to all connected users. This means when someone reacts to a post, everyone online sees the reaction count change immediately. The comments work exactly the same way.

```
// Send reaction updates to all users
socket.on('reaction', (reaction: IReactionDocument) => {
  this.io.emit('update like', reaction);
});
```

37 Ethics

When building this backend system, I felt it was important to work with a strong sense of discipline, responsibility, and good habits. Having a strong work ethic — staying focused, motivated, and making sure to properly finish tasks — was very important during every part of the project. Especially when setting up important systems like login security, real-time chat, and user data management, I couldn't afford to cut corners. Building good habits early, like planning carefully, checking inputs, and protecting data, made a big difference in the quality and reliability of the app.

Besides just working hard, I also thought a lot about the bigger ethical side of the project. Since the app handles sensitive information — like user profiles, private chats, social connections, and maybe even location data in the future — it was critical to handle all of this in a responsible way. Trust from users depends on how well you protect their data behind the scenes, even if they never see the protections directly.

One major ethical focus was making sure privacy and security were built into the app from the beginning. I made sure to use secure login systems, encrypt important information, protect sessions carefully, and use safe tokens. For real-time chat and notifications, I designed everything so private messages would only be seen by the right people and would never be leaked or saved in an unsafe way. Even background tasks, like sending emails or notifications, were built to avoid any risk of exposing private data.

In the end, my approach to ethics was simple: build the system in a way that earns user trust from the start, and stay disciplined in my work so that the final product would be something people could rely on without worrying about their personal information being mishandled.

38 Conclusion

In conclusion, this project gave me a full experience of building a real social media web app from scratch, using technologies like MongoDB, Express.js, React, Node.js, Socket.IO, and Redis. I got the chance to learn not only how to put these tools together but also how to make the system scalable, secure, and real-time. It was a lot of work across both frontend and backend, but seeing everything come together into a real working app made the effort worth it.

Throughout the project, I developed a better understanding of real-time systems, backend architecture, database management, and how to create a smooth user experience. I also realized how important it is to focus on things like security, clean code organization, and planning for future growth right from the beginning.

This project also showed me how important it is to plan properly and stay disciplined throughout the whole process. From setting up the backend environment to building real-time features like messaging and notifications, every small decision added up to a smoother and more stable final product. It wasn't just about coding — it was about thinking like an engineer, making smart choices early, and solving problems before they even happened. The experience I gained from balancing so many different parts of a system — frontend, backend, real-time communication, background jobs, and security — will definitely help me a lot in future projects and in my career as a developer.

Overall, I'm proud of what I built, and I'm excited to keep learning, improving, and applying these skills to even bigger projects in the future.

39 References

[1] Express.js, "Express — Node.js web application framework." [Online]. Available: https://expressjs.com/. Accessed: 31 Aug 2025.

[2] Socket.IO, "Socket.IO Documentation (v4.x)." [Online]. Available: https://socket.io/docs/v4/. Accessed: 31 Aug 2025.

[3] Socket.IO, "Redis adapter (Pub/Sub scaling for multi-node)." [Online]. Available: 4://socket.io/docs/v4/redis-adapter/. Accessed: 31 Aug 2025.

[AR4] BullMQ, "What is BullMQ." [Online]. Available: https://docs.bullmq.io/. Accessed: 31 Aug 2025.

[5] Redis, "Redis Documentation." [Online]. Available: https://redis.io/docs/latest/. Accessed: 31 Aug 2025.

[6] Mongoose, "Mongoose Documentation." [Online]. Available: https://mongoosejs.com/docs/. Accessed: 31 Aug 2025.

[7] Cloudinary, "Upload API Reference." [Online]. Available: https://cloudinary.com/documentation/image_upload_api_reference. Accessed: 31 Aug 2025.

[8] IETF, "RFC 7519: JSON Web Token (JWT)." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519. Accessed: 31 Aug 2025.

[9] Kelektiv, "node.bcrypt.js — bcrypt for Node.js (GitHub)." [Online]. Available: https://github.com/kelektiv/node.bcrypt.js. Accessed: 31 Aug 2025.

[10] Joi, "Joi API Reference." [Online]. Available: https://joi.dev/api/. Accessed: 31 Aug 2025.

[11] Helmet, "Helmet.js Documentation." [Online]. Available: https://helmetjs.github.io/. Accessed: 31 Aug 2025.

[12] SlanaTech, "swagger-stats — API telemetry & APM." [Online]. Available: https://swaggerstats.io/. Accessed: 31 Aug 2025.