Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)

Connect-Online Social Network

By

Ayush Prabhu	**60003220218**
Dewansh Gopani	**60003220115**
Adnan Merchant	**60003220084**
Ayush Parmar	**60003220092**

Under the guidance of

Ms.Prachi Satam

A.Y. 2024 – 2025

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)

Table of Contents

1. Introduction

1.1 Project Overview

This project implements a full-stack social media platform using the MERN stack (MongoDB, Express.js, React.js, Node.js). Users can register and log in, post text and images, follow/unfollow other users, like and “retweet” (repost) content, and comment on posts. A real-time notification system informs users when someone likes or reposts their content or starts following them..

1.2 Purpose and Objectives

- **Enable content creation:** Allow users to publish posts containing text and images.
- **Foster engagement:** Support likes, comments, and reposts to facilitate interaction.
- **Follow system:** Let users follow/unfollow each other to curate personal feeds.
- **Notifications:** Notify users of likes, reposts, comments, and new followers in real time.
- **Responsive UI:** Ensure the interface adapts smoothly across desktop and mobile devices.

1.3 Scope and Limitations

Scope:

- User authentication and profile management.
- Post creation (text + image upload) and feed display.
- Like, comment, and repost functionality.
- Follow/unfollow and personalized feed algorithms.
- Notification centre for user interactions.

Limitations:

- No direct messaging between users.
- Image uploads limited to predefined file types/sizes.
- No hashtag search or trending-topic analysis.
- Notifications stored only in-app (no email/SMS alerts).

1.4 Contribution as a Member of Team

- Designed and implemented both frontend and backend.
- Handled API integration and MongoDB schema modeling.
- Deployed frontend and backend to Render.
- Built React front-end with state management and responsive design..

2. Requirements and Analysis

2.1 Functional Requirements

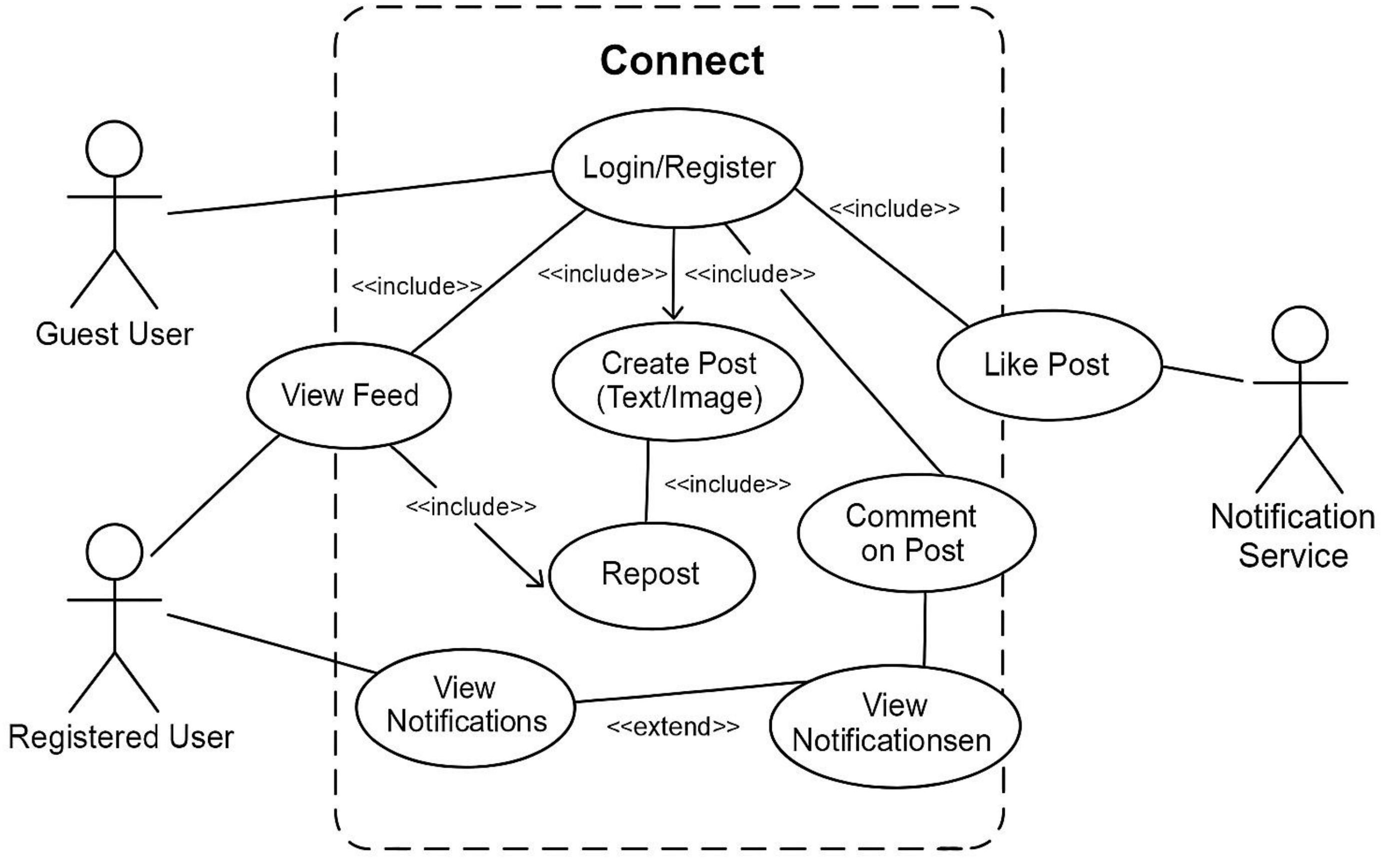
- **User Registration & Login** (JWT-based).
- **Profile Management:** Edit display name, avatar, bio.
- **Post Creation:** Upload text and images.
- **Engagement:** Like, comment, and repost posts.
- **Social Graph:** Follow and unfollow users.
- **Notifications:** Real-time alerts on likes, reposts, follows, comments.
- **Feed:** Personalized timeline ordered by recency & connections.

2.2 Non-Functional Requirements

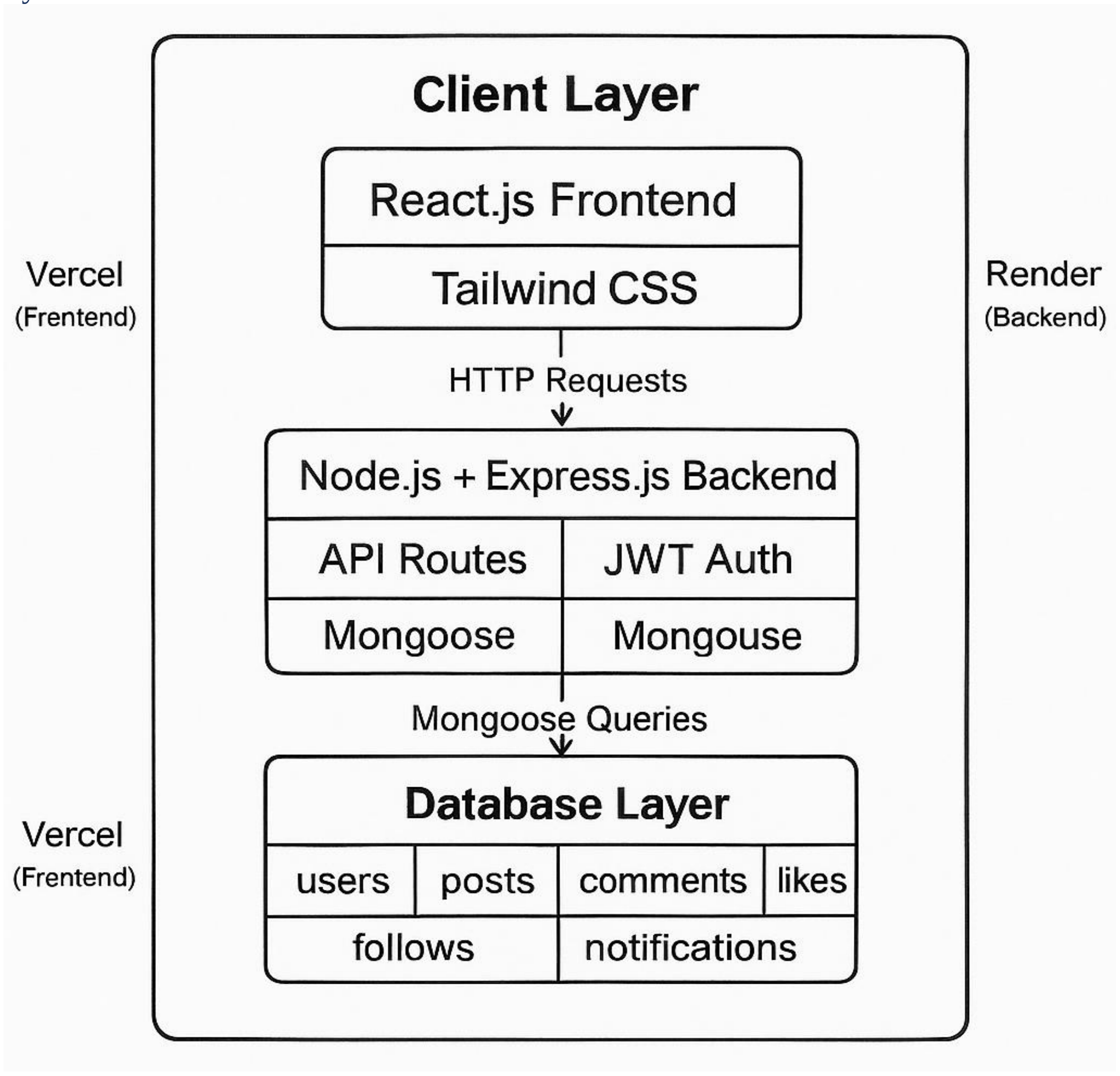
- **Performance:** API latency under 200 ms for common queries.
- **Scalability:** Horizontal scaling of back end and database sharding ready.
- **Security:** Secure password hashing, JWT expiry, CORS policy.
- **Usability:** Intuitive, mobile-first UI with accessible components.
- **Reliability:** Automatic retries for failed image uploads.

2.3 Use Case Diagrams

- **Visitor:** Browse public posts feed.
- **Registered User:**
- Login → Create/Edit Post → View Feed → Engage (like/comment/repost) → View Notifications → Follow/Unfollow.
- **System Admin (future):** Moderate posts, manage user accounts.


Connect
Login/Register
Guest User
<<include>>
<<include>>
<<include>>
<<include>>
View Feed
Create Post (Text/Image)
Like Post
Notification Service
<<include>>
<<include>>
Comment on Post
Repost
Registered User
View Notifications
<<extend>>
View Notificationsen

3. System Architecture



3.1 Database Architecture

MongoDB collections include:

- user (username, fullname, password, email,followers,unfollowers, likedpost,timestamp)
- post (user,test,img,likes,comment)
- notification (from,to,type)

3.2 Interaction Flow between Frontend and Backend

- Frontend sends HTTP requests (Axios/Fetch) to backend routes.
- Backend APIs interact with MongoDB and return responses.
- JWTs are used to secure protected routes.

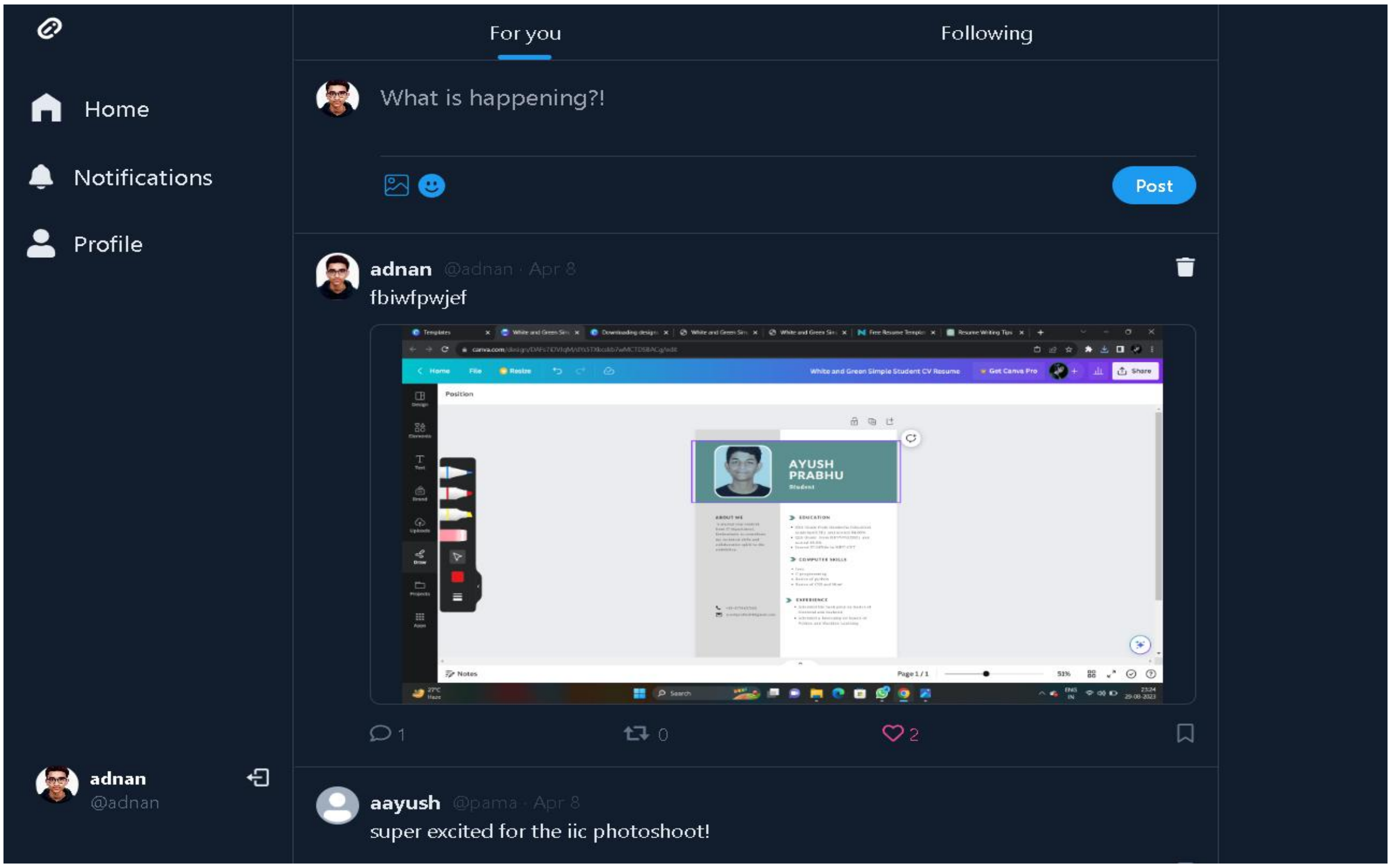
3.3 Tools and Frameworks

- **Frontend**: React.js, TailwindCSS,
- **Backend**: Node.js, Express.js, MongoDB, Mongoose

- **Authentication**: JWT
- **Deployment**: Render(Frontend), Render (Backend)

4. Frontend Development

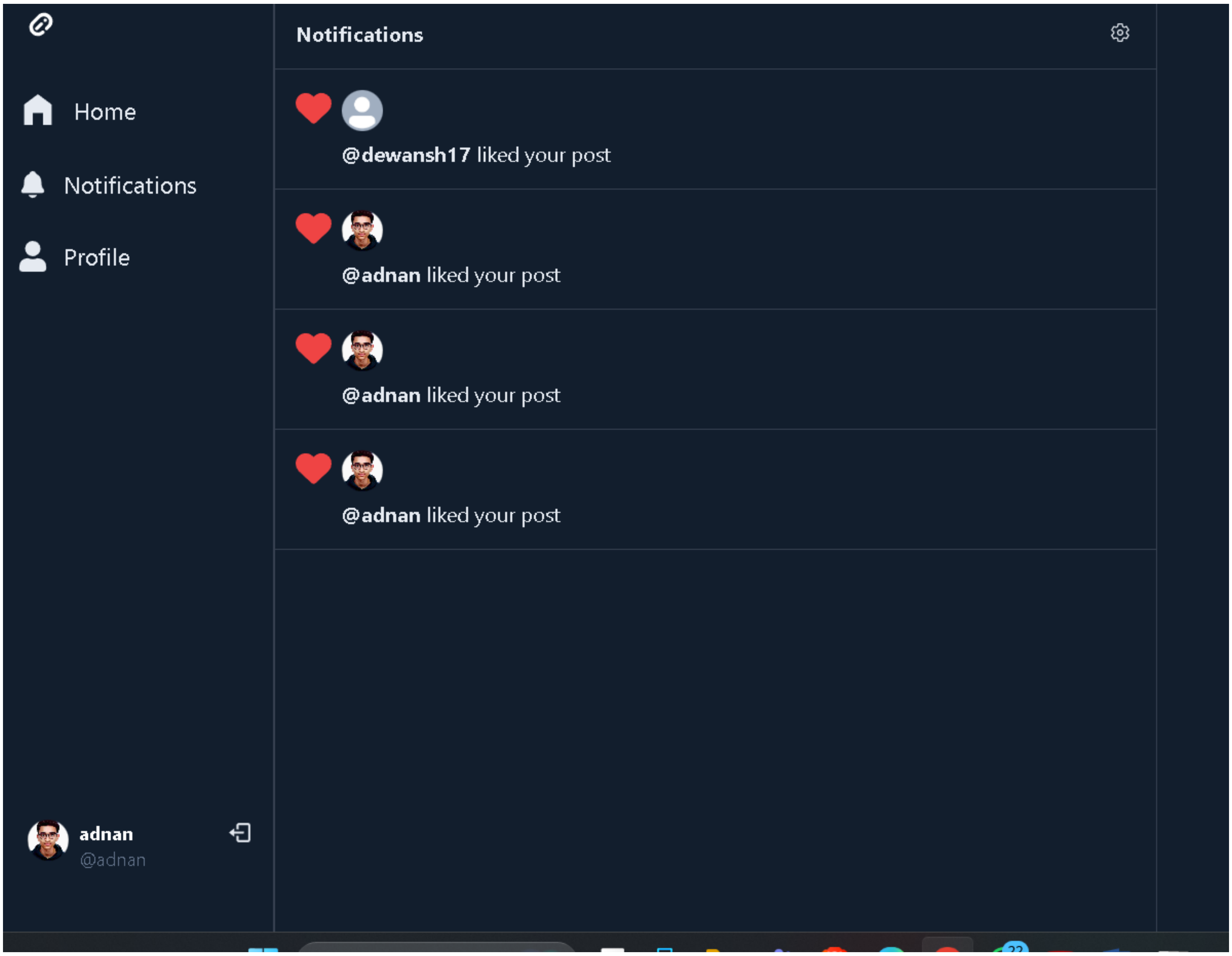
4.1 Design and Layout (UI/UX)



fbiwfpwjef

super excited for the iic photoshoot!

Notification



Notifications

@dewansh17 liked your post

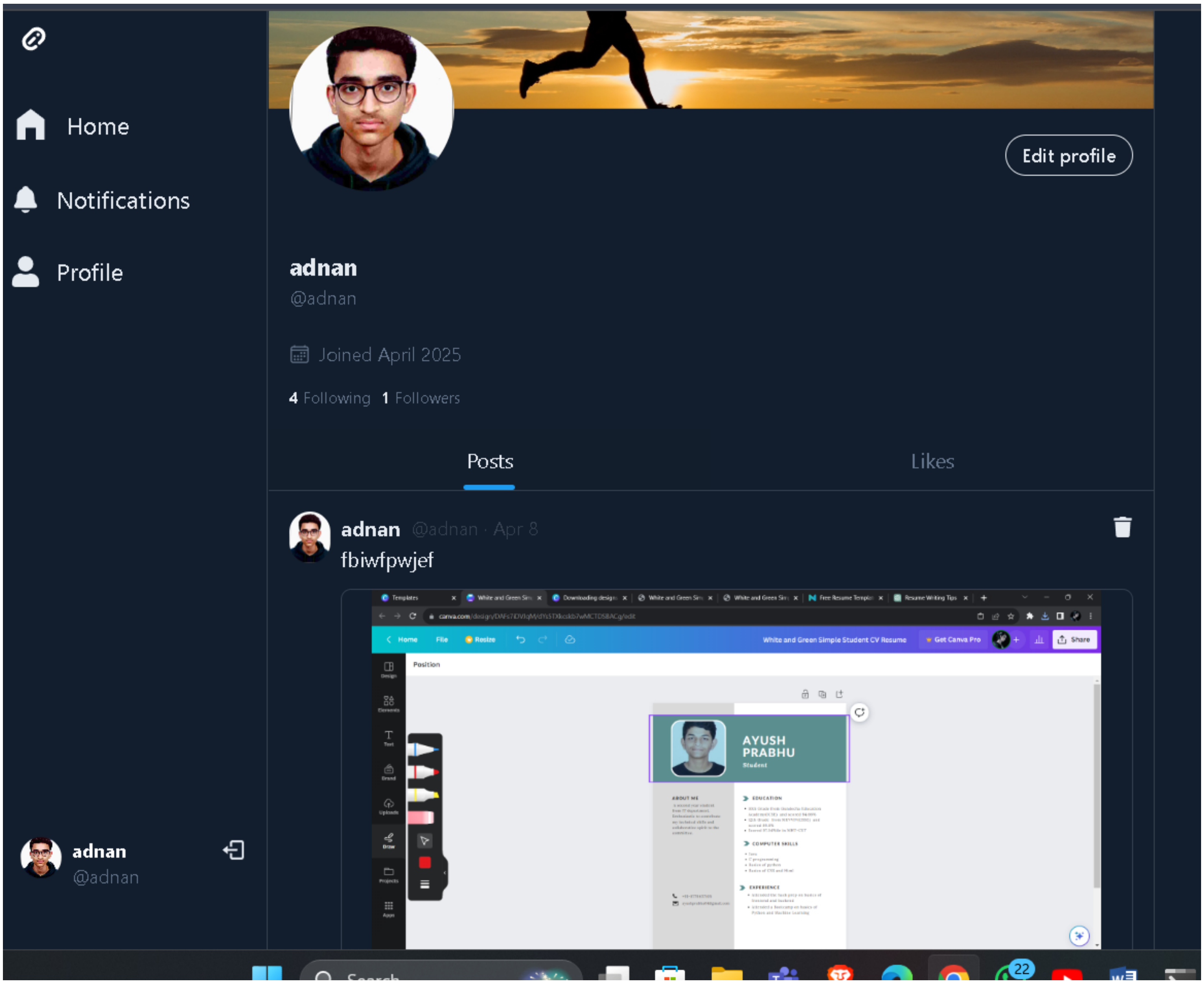
@adnan liked your post

@adnan liked your post

@adnan liked your post

`

Profile



adnan
@adnan

Joined April 2025

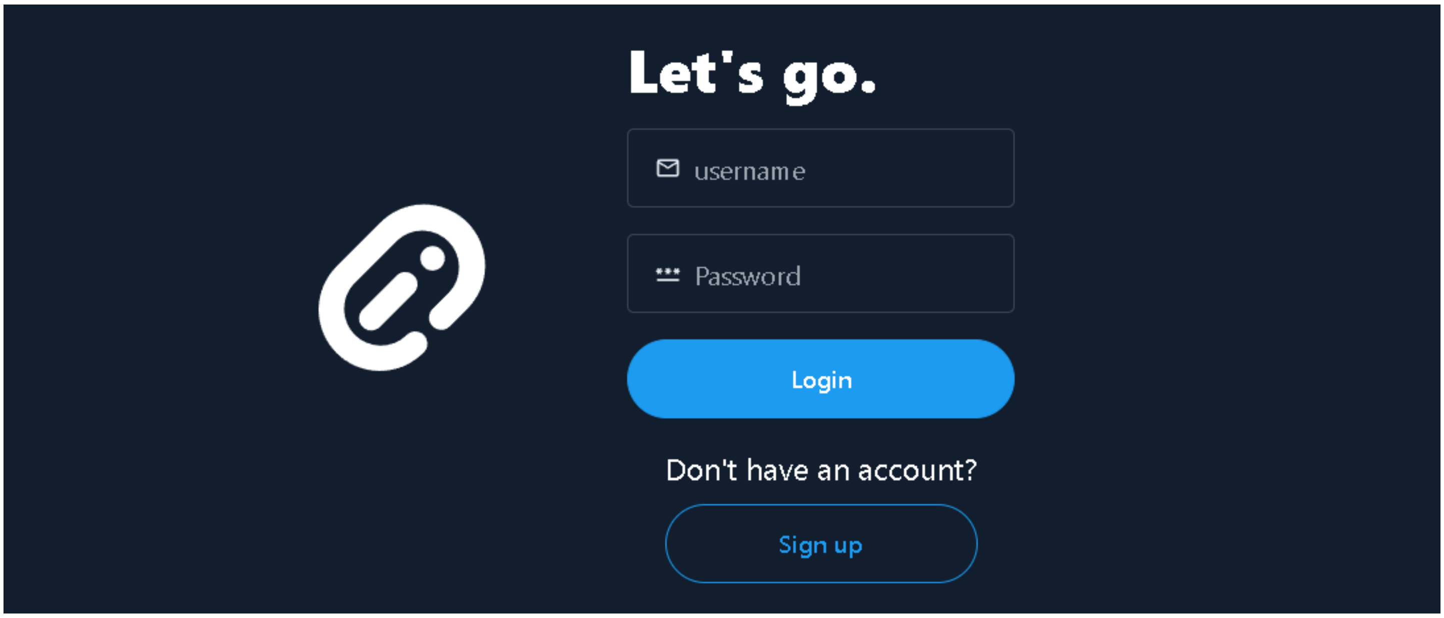
4 Following 1 Followers

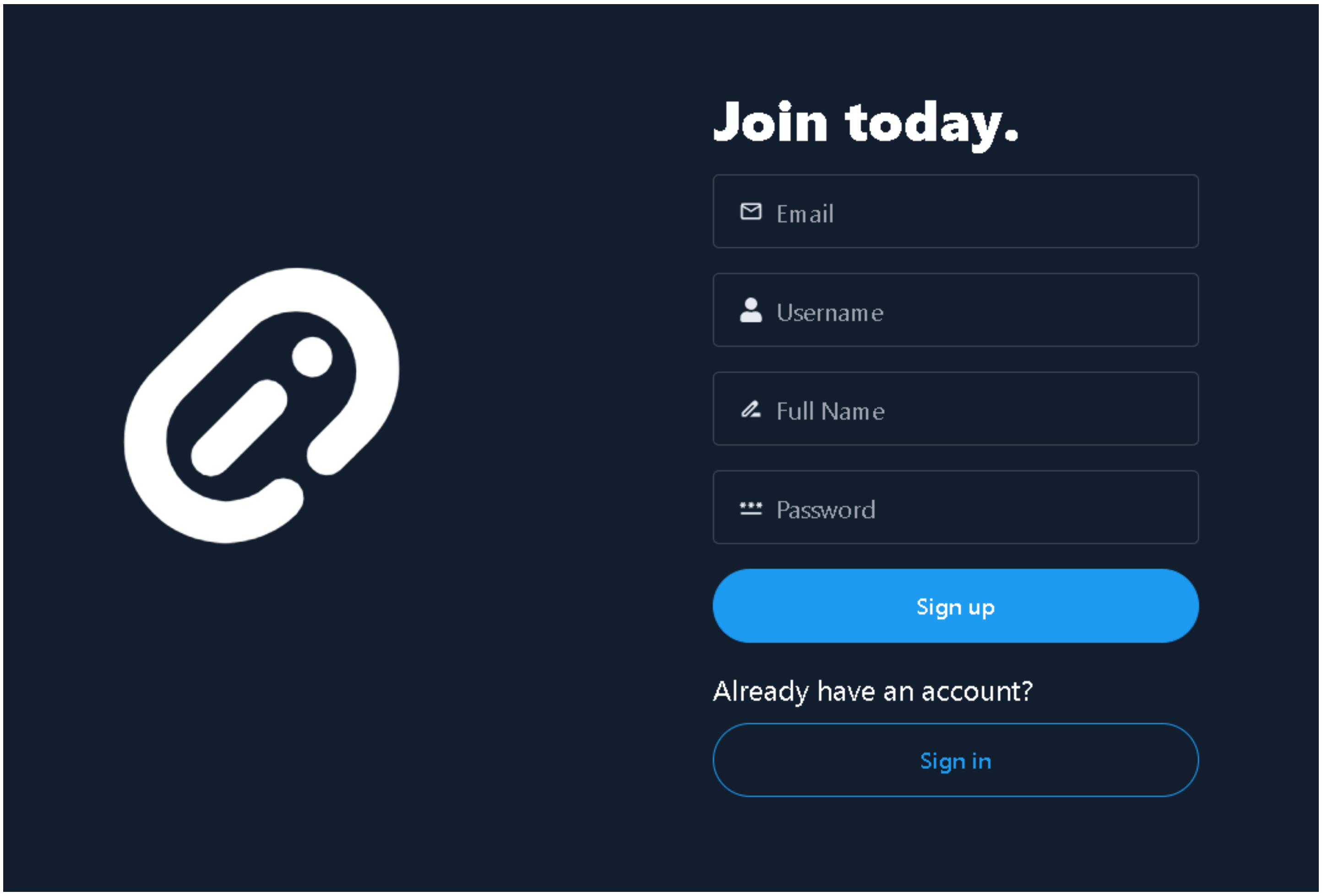
adnan @adnan · Apr 8
fbiwfpwjef

adnan
@adnan

4.2 Components and Pages

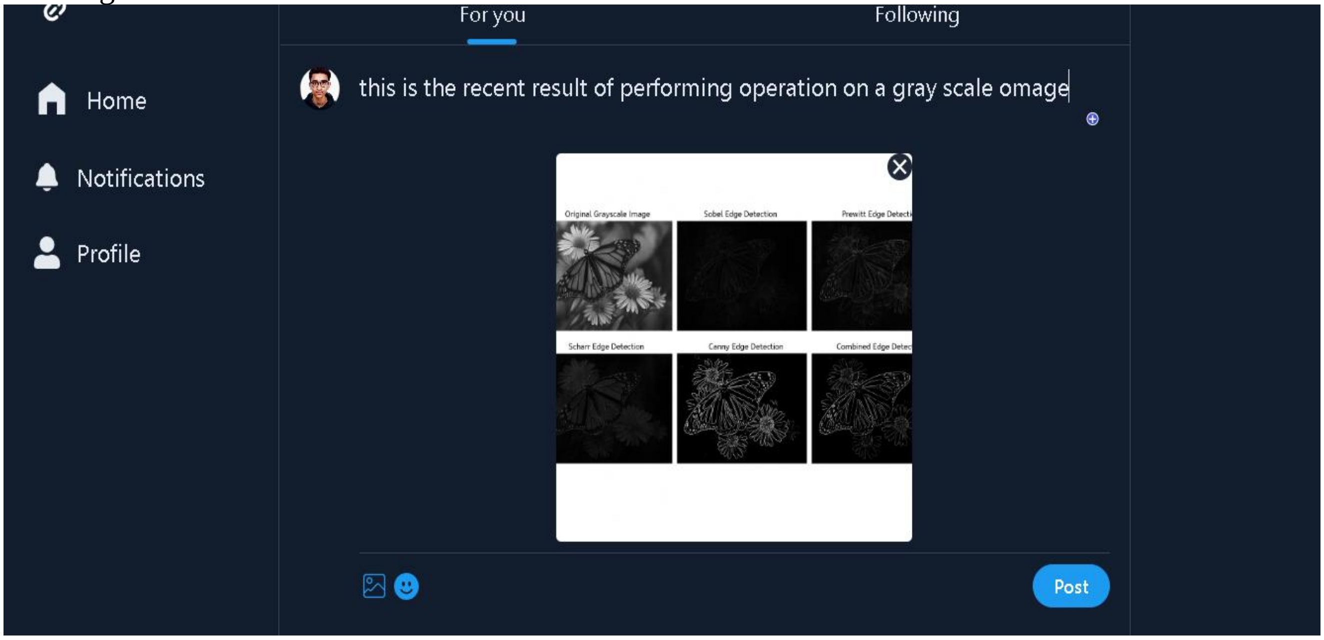
- Login/Register page





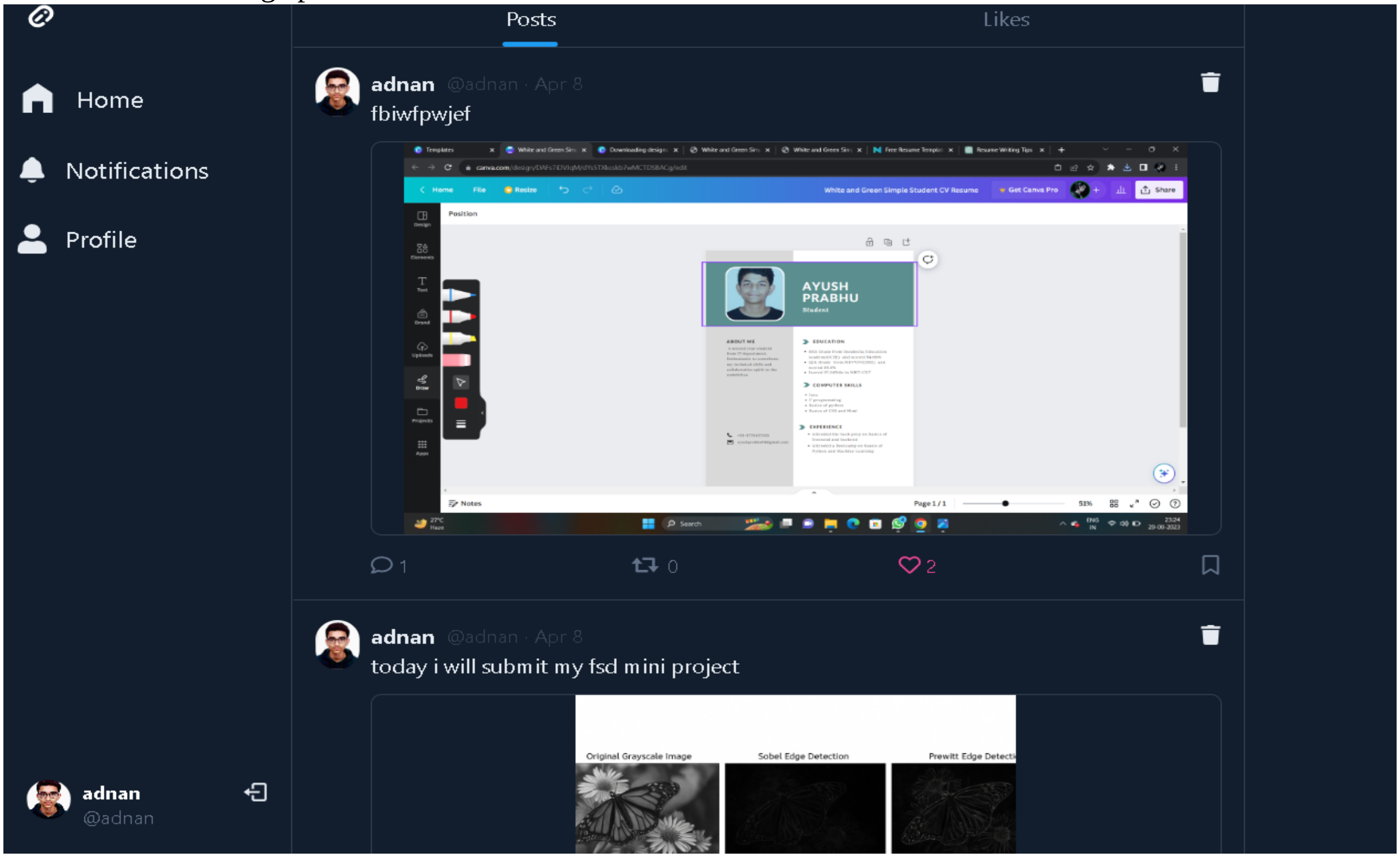
Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)

- Post Page



this is the recent result of performing operation on a gray scale omage

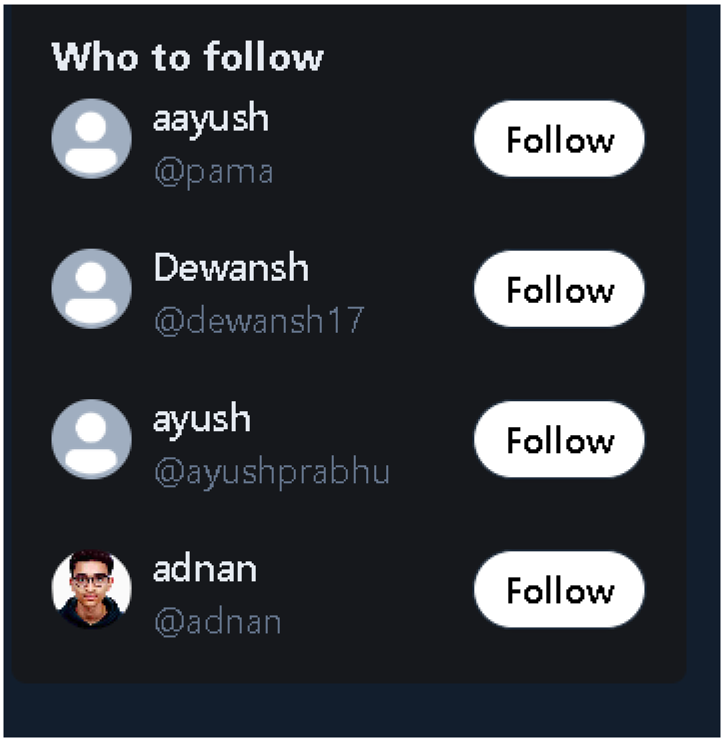
-
- Profile Page post



adnan @adnan · Apr 8
fbiwfpwjef

adnan @adnan · Apr 8
today i will submit my fsd mini project

- Personalised User recommendation



4.3 Error Handling and Validation

- Client-side validation using regex and conditionals.
- Toast messages for success/error status.
- Display messages for invalid login or failed uploads.

5. Backend Development

5.1 Backend Technologies Used

- Node.js with Express.js
- MongoDB with Mongoose
- JSON Web Tokens (JWT)

5.2 RESTful API Development

- `POST /api/auth/login`
- /api/notificarion – view notification
- /api/profile/username– profile page

5.3 Database Schema and Models

- user (username, fullname, password, email,followers,unfollowers, likedpost,timestamp)
- post (user,test,img,likes,comment)
- notification (from,to,type)

5.4 Server Configuration and Setup

- .env file for environment variables
- Configured CORS and middleware in server.js
- MongoDB connected using Mongoose

5.5 Authentication and Authorization

- JWT-based login, with role-based access control
- Middleware to check token and user role

5.6 Data Validation and Error Handling

- Backend input validation using custom middleware
- Try-catch blocks and Express error handling middleware

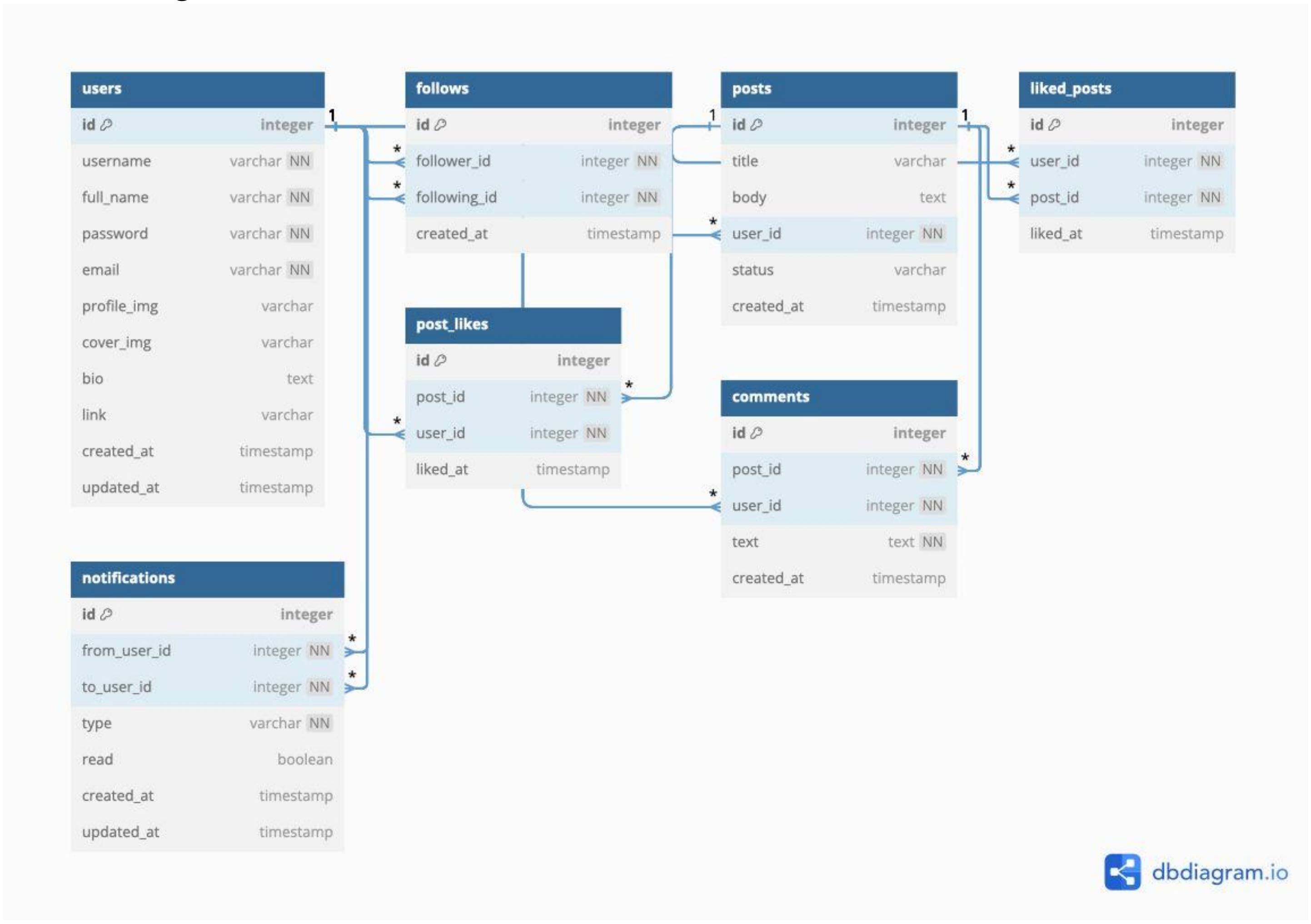
6. Database Design

6.1 Schema Design

```
// models/User.js
import mongoose from "mongoose";

const userSchema = new mongoose.Schema(
	{
		username: {
			type: String,
			required: true,
			unique: true,
		},
		fullName: {
			type: String,
			required: true,
		},
		password: {
			type: String,
			required: true,
			minLength: 6,
		},
		email: {
			type: String,
			required: true,
			unique: true,
		},
		followers: [
			{
				type: mongoose.Schema.Types.ObjectId,
				ref: "User",
				default: [],
			},
		],
		following: [
			{
				type: mongoose.Schema.Types.ObjectId,
				ref: "User",
				default: [],
			},
		],
		profileImg: {
			type: String,
			default: "",
		},
		coverImg: {
```

```
                        type: String,
                        default: "",
                },
                bio: {
                        type: String,
                        default: "",
                },

                link: {
                        type: String,
                        default: "",
                },
                likedPosts: [
                        {
                                type: mongoose.Schema.Types.ObjectId,
                                ref: "Post",
                                default: [],
                        },
                ],
        },
        { timestamps: true }
);

const User = mongoose.model("User", userSchema);

export default User;

  // models/Post.js

  import mongoose from "mongoose";

  const postSchema = new mongoose.Schema(
        {
                user: {
                        type: mongoose.Schema.Types.ObjectId,
                        ref: "User",
                        required: true,
                },
                text: {
                        type: String,
                },
                img: {
                        type: String,
                },
                likes: [
                        {
                                type: mongoose.Schema.Types.ObjectId,
					ref: "User",
				},
			],
			comments: [
				{
					text: {
						type: String,
						required: true,
					},
					user: {
						type: mongoose.Schema.Types.ObjectId,
						ref: "User",
						required: true,
					},
				},
			],
	},
	{ timestamps: true }
);

const Post = mongoose.model("Post", postSchema);

export default Post;

// models/notification.js

import mongoose from "mongoose";

const notificationSchema = new mongoose.Schema(
	{
			from: {
				type: mongoose.Schema.Types.ObjectId,
				ref: "User",
				required: true,
			},
			to: {
				type: mongoose.Schema.Types.ObjectId,
				ref: "User",
				required: true,
			},
			type: {
				type: String,
				required: true,
				enum: ["follow", "like"],
			},
			read: {
				type: Boolean,
                    default: false,
            },
    },
    { timestamps: true }
);

const Notification = mongoose.model("Notification", notificationSchema);

export default Notification;
```

6.2 ER Diagram



6.3 Data Integrity and Validation

- Unique constraints on email
- Status enum for booking requests
- Role-based constraints enforced via middleware

7. Integration of Frontend and Backend

7.1 API Integration Overview

- Role tokens stored in local storage for auth.
- Protected routes render conditionally based on login state.

7.2 State Management (if applicable)

- React useState and useEffect hooks
- Context API used to share auth/user info globally

7.3 Error Handling During Integration

- Try-catch blocks and error boundaries in React.
- Backend errors passed via JSON response and shown in frontend alerts

8. Testing

8.1 Unit Testing (Frontend and Backend)

- Basic testing using console and Postman for backend APIs.
- Component testing using manual validation in browser.

8.2 Integration Testing

- End-to-end booking flow tested from form to approval.
- Timetable upload tested for various Excel formats.

8.3 Functional Testing

- All roles tested with expected and edge-case inputs.
- Booking rejection and update scenarios verified.

9. Deployment

9.1 Deployment Strategy

- Frontend deployed on Vercel (CI/CD enabled with GitHub)
- Backend deployed on Render with auto-redeploy on commit

9.2 Setting Up the Server

- Render backend: Configured start command and environment variables
- MongoDB Atlas used as cloud DB

9.3 Domain Name and Hosting

- Vercel auto-generated URL used (can map custom domain if needed)
- Render backend served via HTTPS endpoint

10. Conclusion

10.1 Project Summary
Built a scalable, real-time social media platform with core features: posting, liking, commenting, following, and notifications..
10.2 Challenges Faced

- Handling concurrent updates to likes/reposts.
- Efficiently pushing real-time notifications at scale.
- Designing a clean, responsive UI.
- API integration with proper error handling.

10.3 Future Enhancements and Improvements

- Direct messaging between users.
- Hashtag and search functionality.
- Email/SMS notification integration.
- Analytics dashboard (post reach, engagement metrics).

.

10.4 Learning Outcomes

- Deepened understanding of MERN architecture.
- Experience with real-time WebSocket integrations.
- Best practices in REST API design and secure authentication.
- Improved understanding of role-based authentication and backend integration.

11. References

- MongoDB & Mongoose documentation
- Express.js official guides
- React.js and Tailwind CSS docs
- Socket.io real-time communication guide