

Twitter Thought Note

Introduction:

This is my final project in the full stack specialization. It is a full-blown social media application(twitter-clone) with both the front end (User Interface, UI) and back end (security, database, and APIs).

My flow of creating this application was building the frontend first. And then moving to the backend part. Although it was suggested otherwise, building frontend first made me realize that I might need to add an extra field or two in Schema and create a few different routes to make the app work finely.

I'll explain the flow of my application building here. And will also add GitHub and deployment(I'll be using render for deployment) links for hosting my application.

Github link: <https://github.com/Trupti0406/twitter-clone>

Deployed link(live demo): <https://twitter-by-trupti.onrender.com>

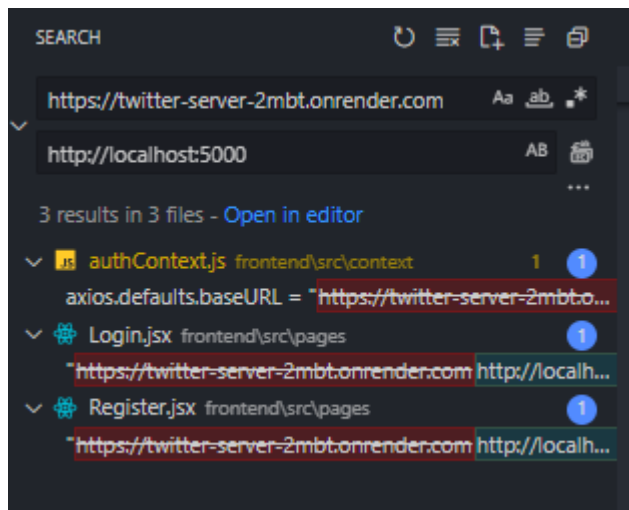
Important note for the evaluator:

For deployment purposes, I have changed the url in frontend to my render server url. Please change it back to “<http://localhost:5000>” before starting the project:

Steps:

In GitHub search, simply search for

<https://twitter-server-2mbt.onrender.com> change and replace it with <http://localhost:5000> like this:



For reference I also commented on the original url in all three files, you can access them by simply uncommenting them.

1) authContext.js:

```
const registerRequest = async (e) => {
  e.preventDefault();
  const { data } = await axios.post(
    // "http://localhost:5000/auth/register",
    "https://twitter-server-2mbt.onrender.com/auth/register",
    registerDetails
  );
}
```

2) Login.jsx

```
const loginRequest = async (e) => {
  e.preventDefault();
  const { data } = await axios.post(
    // "http://localhost:5000/auth/login",
    "https://twitter-server-2mbt.onrender.com/auth/login",
    loginDetails
  );
}
```

3) Register.jsx:

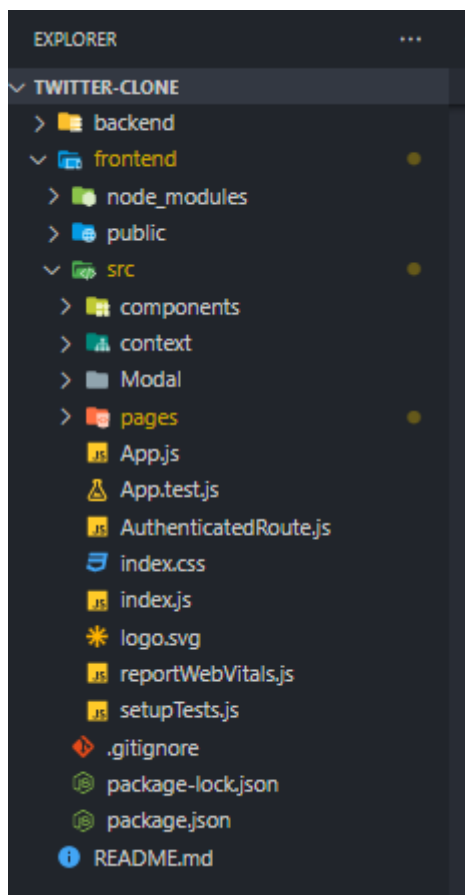
```
const registerRequest = async (e) => {
  e.preventDefault();
  const { data } = await axios.post(
    // "http://localhost:5000/auth/register",
    "https://twitter-server-2mbt.onrender.com/auth/register",
    registerDetails
  );
}
```

Part 1 – Frontend

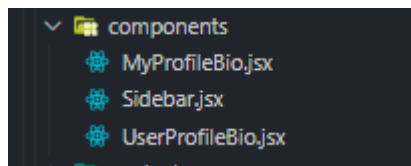
Tech Stack Used:

- Reactjs
- Bootstrap (UI library)
- react-router-dom (routing in UI)
- react-toastify (for notifications)
- axios (call APIs)
- fontawesome (for icons)
- Moment (to show the time of tweet creation and dates in a better format)

Folder and File Structure:



I have added 3 components:



1) MyProfileBio.jsx:

Here in this component, I have handled the code related to the Bio of logged-in users, which allows you to add profile photos and edit profile details.

2) Sidebar.jsx:

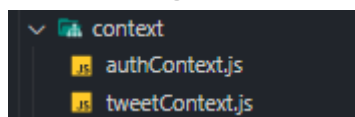
This is the sideBar component, which shows the Home component on which we will render all the tweets. And a button on the top which will open a new modal to create a tweet.

The second item on the sidebar is Profile, this will lead us to our own/logged-in user's profile page, which will further show all the detailed activity of that particular user. The next item is Logout, By clicking on this, the user will be logged out and redirected to the login page.

3) UserProfileBio.jsx:

This page contains the bio and activity of other users that are using the app. No other user would be allowed to edit or add tweets to someone else's account.

I am using React context\

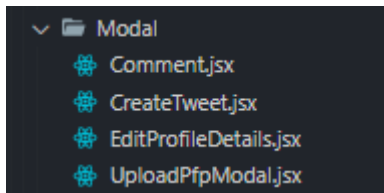


I have used react context in my application, and I divided the context code into two distinct files, authContext(for auth-related code) and tweetContext(for tweet-related code).

authContext.js contains the authentication provider component, and tweetContext.js is used to provide tweet-related data to the component, both of these would be used in our index.js file to wrap our app in them.

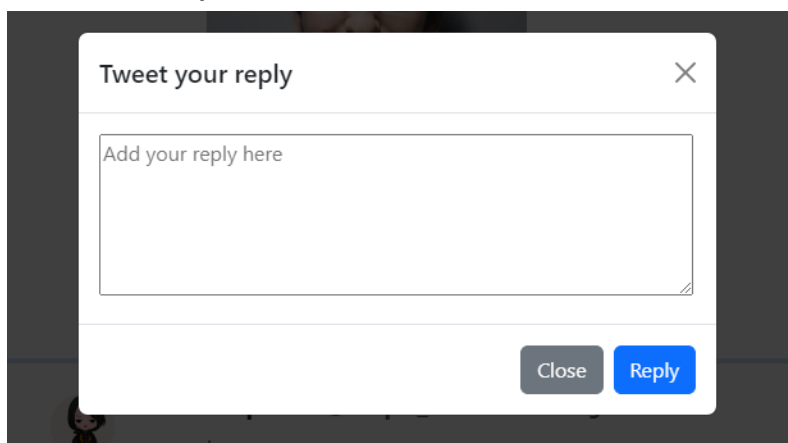
Modal Folder:

In the Modals folder, I created 4 modals in total.



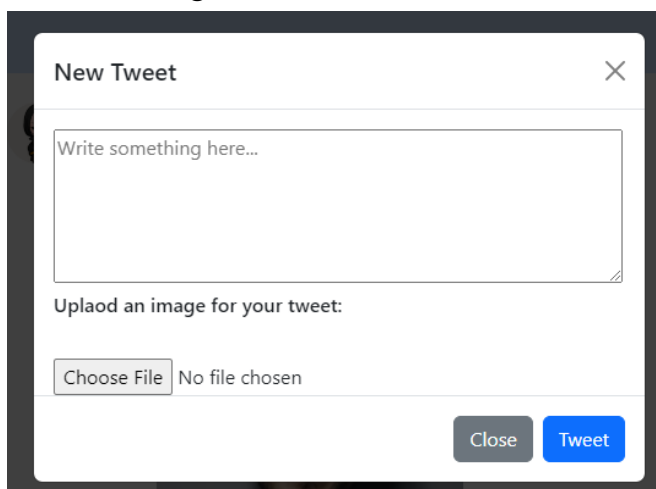
1) **Comment.jsx**

I created Bootstrap modals, creation of modals through Bootstrap is very easy. This component is responsible for adding a comment. Whenever we click on the comment icon in the tweet card, this modal will open, this is how it looks like:



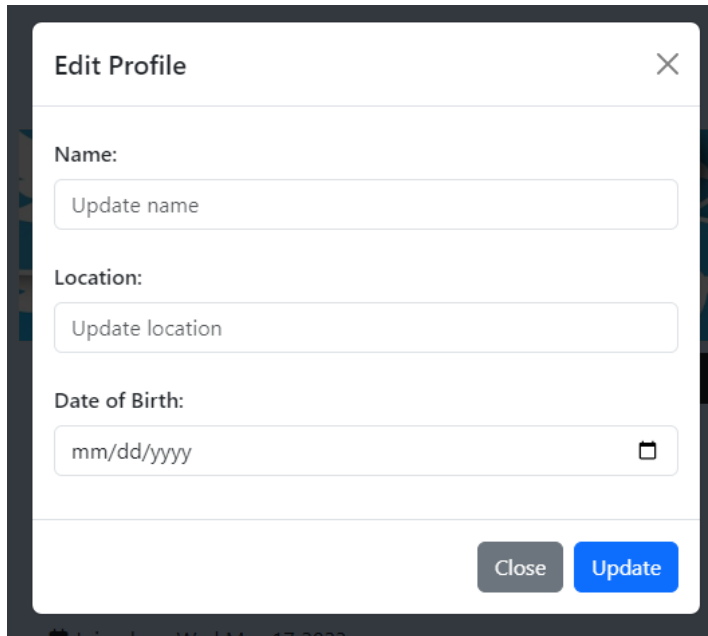
2) **CreateTweet.jsx:**

This modal will allow logged-in users to create a new tweet. On our feed page a button named "Tweet" is available. Once we click on that button, this modal will open, where we can add text content as well as images to the tweet. Tweet creation modal looks like this:



3) EditProfileDetails.jsx:

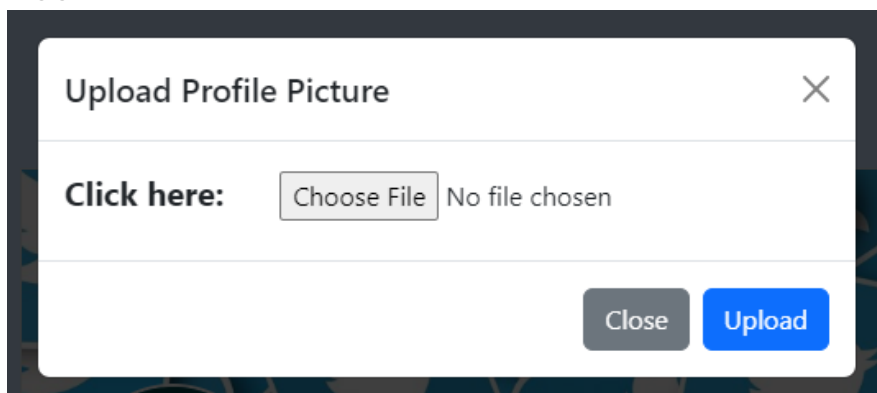
This modal is responsible for editing the profile details of a user. Only logged-in users, i.e. admin users will be allowed the edit the profile details like this, the button to open this modal would be available on the profile page of the logged-in user. And the modal looks like this:



The screenshot shows a modal titled "Edit Profile" with a close button (X) in the top right corner. The modal contains three input fields: "Name:" with a placeholder "Update name", "Location:" with a placeholder "Update location", and "Date of Birth:" with a placeholder "mm/dd/yyyy" and a calendar icon. At the bottom right, there are two buttons: "Close" (grey) and "Update" (blue).

4) UploadPfpModal.jsx:

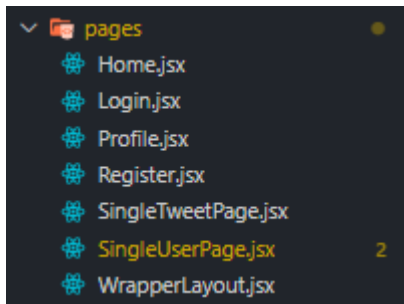
This modal opens when we click on the upload profile picture button on the profile page, It looks like this. And allows the user to upload and changes the profile picture of the authenticated logged-in user, It looks like this:



The screenshot shows a modal titled "Upload Profile Picture" with a close button (X) in the top right corner. The modal contains a "Click here:" label, a "Choose File" button, and the text "No file chosen". At the bottom right, there are two buttons: "Close" (grey) and "Upload" (blue).

Pages/screens:

I created 6 pages in total and then added 1 more extra page (WrapperLayout.jsx) as suggested by the mentor in a doubt session.



1) WrapperLayout.jsx:

While rendering components, especially when you have a sidebar or navbar component in your app this method works like a charm. It can be used when you have nested routes in your app.

Reference:

<https://medium.com/age-of-awareness/amazing-new-stuff-in-react-router-v6-895ba3fab6af>

```
import React from "react";
import { Outlet } from "react-router-dom";
import Sidebar from "../components/Sidebar";

const WrapperLayout = () => {
  return (
    <>
      <Sidebar />
      { /* ! profile section */ }
      <Outlet />
    </>
  );
};

export default WrapperLayout;
```

First I'll let you know about the wrapperLayout component. Since this explains the structuring of other pages:

The **outlet** component from react-router-dom is used as a placeholder to render the content of the nested routes within the WrapperLayout component.

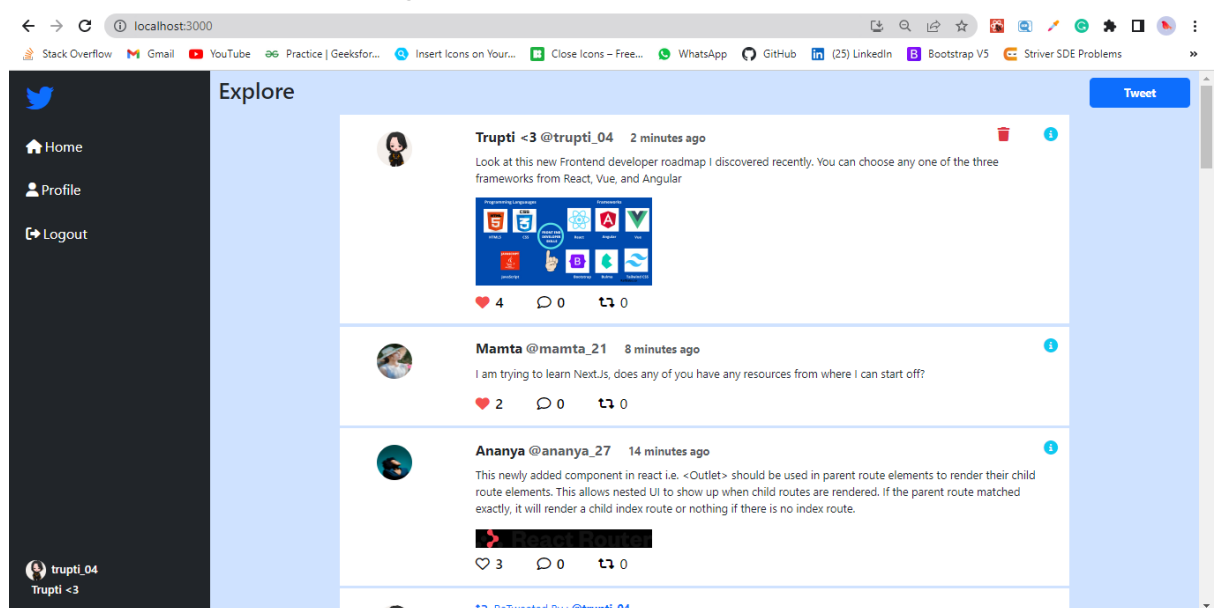
The Outlet component acts as a marker or a slot where the content of the matched nested route will be rendered. It is typically used within the parent component of a nested route to define the location where the nested route's content should appear.

In this case, the Sidebar component is rendered first, followed by the Outlet component. The Outlet component will render the content of the matched nested route within its location in the component hierarchy. By using the Outlet component, the WrapperLayout component serves as a layout container that provides a consistent structure or layout for the content rendered within its nested routes. The Sidebar component is rendered consistently for all the routes, while the specific content of each route is rendered within the Outlet component, allowing for modular and organized routing in the application.

2) Home.jsx:

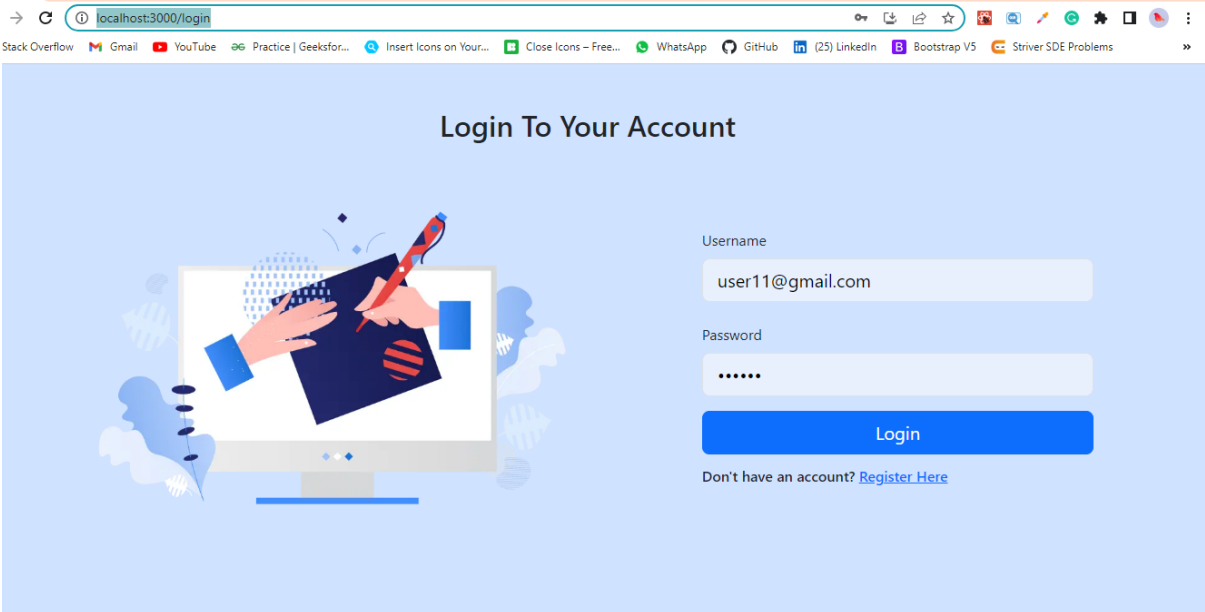
This page is basically our index page, after logging in user will be redirected to this page. All tweets along with replies will be available on this page, Home page will also have a sidebar and a button to add tweets. The tweet card is fully functioning.

This is how the homepage looks:



3) Login.jsx

The login page is the basic login page used in every MERN application built using JWT. Here's how it looks in my app:



localhost:3000/login

Stack Overflow Gmail YouTube Practice | Geeksfor... Insert Icons on Your... Close Icons - Free... WhatsApp GitHub (25) LinkedIn Bootstrap V5 Striver SDE Problems

Login To Your Account

Username

user11@gmail.com

Password

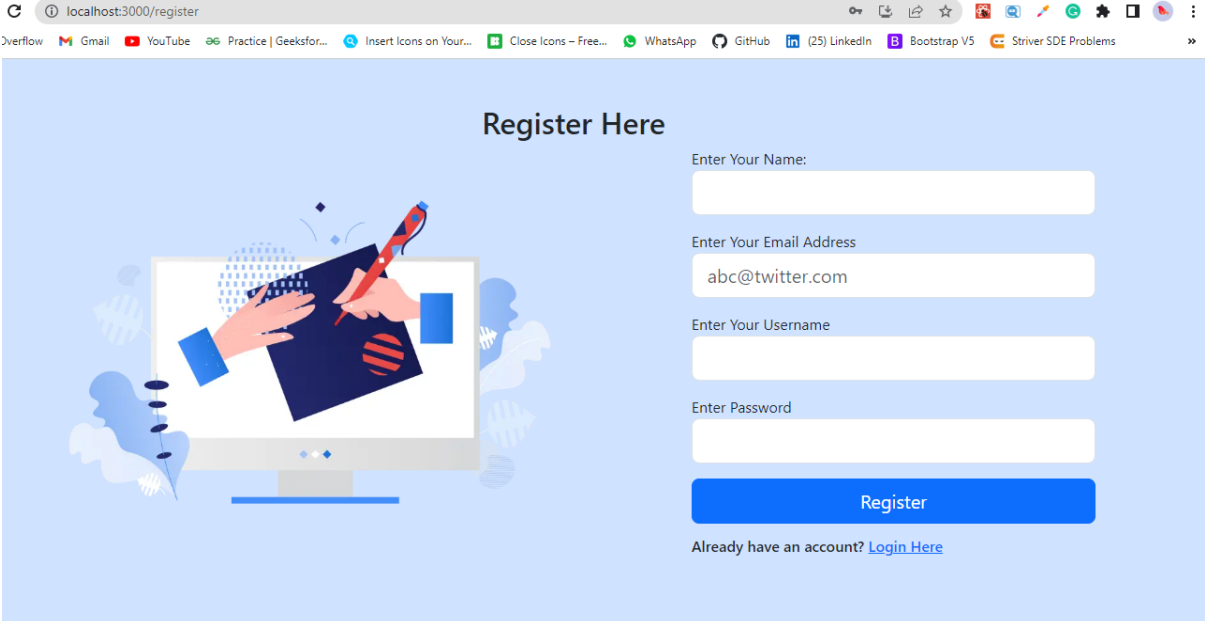
.....

Login

Don't have an account? [Register Here](#)

4) Register.jsx:

This is how my registration of user page looks:



localhost:3000/register

Stack Overflow Gmail YouTube Practice | Geeksfor... Insert Icons on Your... Close Icons - Free... WhatsApp GitHub (25) LinkedIn Bootstrap V5 Striver SDE Problems

Register Here

Enter Your Name:

Enter Your Email Address

abc@twitter.com

Enter Your Username

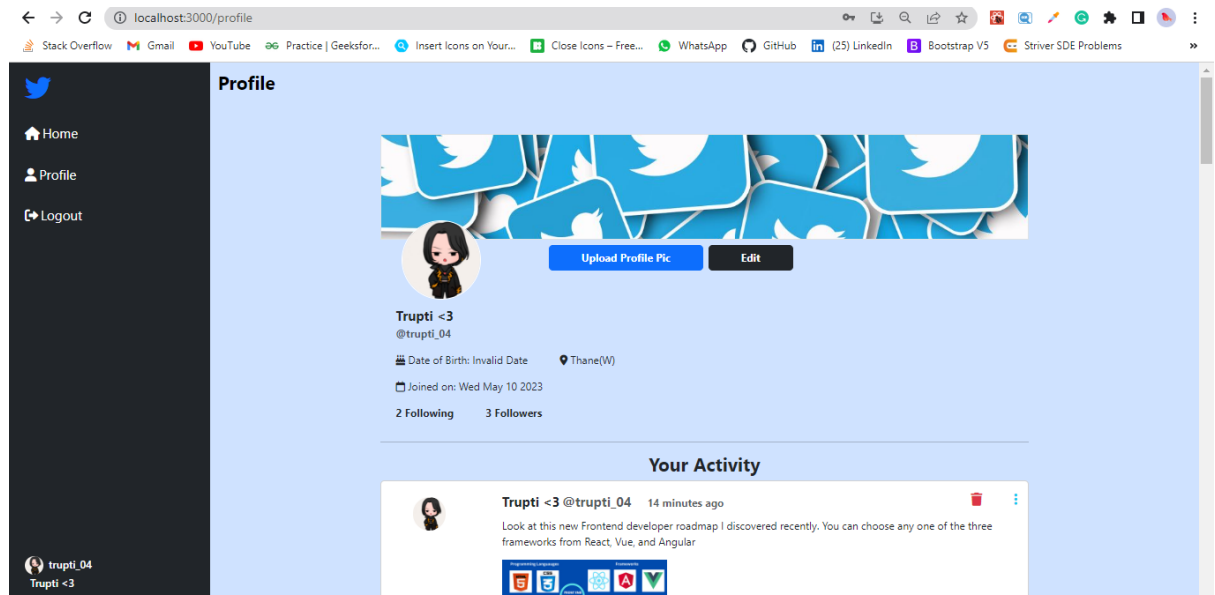
Enter Password

Register

Already have an account? [Login Here](#)

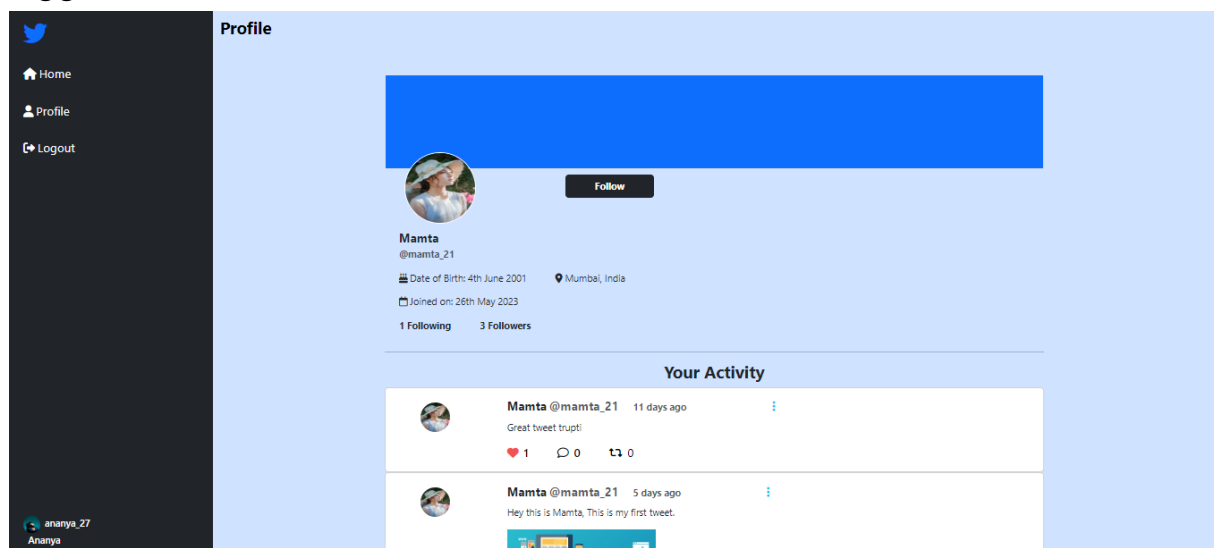
5) Profile.jsx:

This is how my profile picture looks, instead of follow/unfollow button here we have an Upload profile pic button and an Edit profile details button:



6) SingleUserPage.jsx:

This is how singleUserPage appears. It has all the tweets created by that user and It has a follow/unfollow button which works like a toggle button:



7) SingleTweetPage.jsx:

This page shows the details of the single tweet and all the replies to it. Including the nested replies, this is how it looks:



App.js component:

In my app.js component, I have included all routes where login and register are separated and the rest of the routes are nested, for <Outlet/> component.

Index.js file:

In my index.js file, I've wrapped my <App/> component in AuthProvider, TweetProvider, and BrowserRouter. So that our app can access the functionalities of all these.

Index.css:

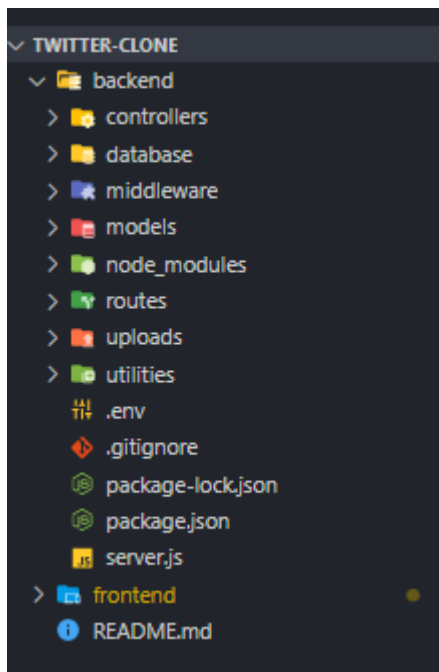
I have tried using Bootstrap as much as possible, but I've used some CSS in the like/ dislike functionality and while creating the profile bio component. I didn't really find the need to create separate CSS files and increase the import and render time for a few lines of styling. So I added them in the root CSS file i.e. index.css.

Part 2 – Backend

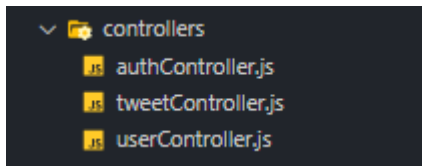
Tech Stack Used:

- Node.js (main backend language)
- Express.js (framework based on nodejs to create REST APIs)
- Mongoose (to connect MongoDB with express.js)
- bcrypt (to encrypt password when stored in DB)
- multer (to upload images)
- Sharp (for image processing)
- Uuid (to generate a unique name for uploaded photo urls)
- JWT (for authentication)
- express-fileupload (for file upload)
- Cloudinary (to upload images to cloud)
- dotenv (to access variables in .env file)

Folder and file structure:



Controllers:



This folder has three main controller files, in `authController.js` all the functions related to auth API are created and exported to further use in routes. In auth controller there are only two functions one for login and register.

Similarly, `tweetController` and `UserController` handle all the functions used in API requests of the corresponding routes.

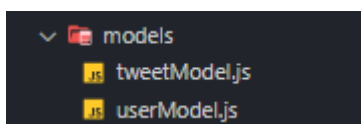
Database:

In the database folder, I have one single file `connect.js`, in which I have configured the MongoDB connection to my server. Which is further used in my `server.js` file.

Middleware:

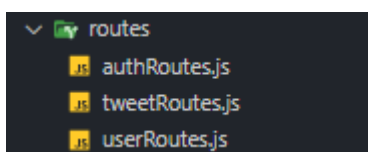
In the middleware folder, I have `auth.js` file. This file is used for authentication purposes, by passing this middleware function in request routes we make sure that only authenticated users can access certain routes.

Models:



These two files define the schema for the user and model. I built them as instructed in the problem statement, but I added a few fields in the tweet schema for convenience.

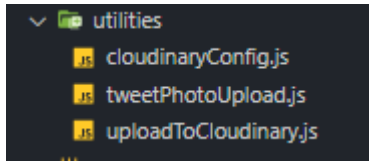
Routes:



Here I have configured all the routes with POST, GET, PUT, and DELETE requests using middleware and controller functions.

Utilities:

This is rather a new folder, here I have configured the functions required to upload the image to Cloudinary and Multer.



cloudinaryconfig.js file has the configuration for cloudinary, tweetPhotoUpload.js contains all the necessary multer-related code for uploading and showing the image on our react app. uploadToCloudinary.js takes care of the uploading of files to Cloudinary storage.

Uploads:

This folder is used as a path provided in multer configuration to store/access images. Later we delete them using the file module in nodeJs

Then we have .env file where all the secret variables are stored like this

```
CLOUD_NAME =  
CLOUD_API_KEY =  
CLOUD_API_SECRET =  
MONGODB_URI =  
JWT_SECRET =
```

Here's a brief description of the functionalities that I have built:

1. Setup project and create a server on PORT

- Start by creating a Nodejs project and installing ExpressJs init.
- Create an index.js file and listen to the express server on any port.

2. Connect the app with MongoDB

- Installed Mongoose and connected the app with the online MongoDB atlas.

- Showed a `console.log("DB connected")`, when MongoDB is connected to indicate and confirm that DB is connected successfully.

3. Creating User Schema:

- Before implementing user-related APIs I implemented created a User MongoDB schema to define how the data will look like.
- I used the `{ timestamp: true }` feature of Mongoose which will store the `createdAt` and `updatedAt` dates of every document. Which I have used in frontend.

4. Creating Tweet Schema:

Just like User Schema, I created the tweet schema. Here I added a few fields for my convenience.

5. Creating Auth-related APIs:

- The prefix for auth API is `/auth` and then every route is following the REST API suffix. Eg. the register route in auth will be `/auth/register` and the complete route with `/API` at the front will be `API/auth/register`
- In auth entity, we will have two APIs `/auth/register` and `/auth/login`. register API will create a new user with input from the user and login will generate a JWT token and will help the user to authenticate and log in.

6. Creating tweet-related APIs:

Below are the functionalities we will cover in tweet API

- a. Create a tweet
- b. Like a tweet
- c. Dislike a tweet
- d. Reply on a tweet
- e. Get a single tweet with details
- f. Get all tweet details
- g. Delete a tweet
- h. Retweet

To test all the APIs I used the built-in VS Code tool of Thunder Client.

Problems I faced while creating this application:

- **Authentication and Authorization:** I faced challenges in implementing a secure and reliable authentication system, including user registration, login, and managing access control to different resources.
- **Data Modeling and Database Design:** I encountered difficulties in designing the database schema and ensuring proper data modeling, especially when dealing with complex relationships between entities such as users, tweets, comments, and likes.
- **API Development and Documentation:** Developing robust and well-documented APIs that handle various CRUD operations for different entities was time-consuming and prone to errors.
- **File Upload and Image Processing:** Integrating file upload functionality, such as uploading profile pictures or tweet images, and handling image processing implementation.
- **Real-time Updates and Notifications:** Implementing real-time updates for features like live feed, notifications, and deletion created a lot of issues.
- **Handling Concurrent Requests and Performance Optimization:** Ensuring the application could handle a large number of concurrent requests and optimizing performance by implementing caching mechanisms, query optimizations, and efficient data fetching strategies was a challenging task.
- **UI/UX Design and Responsiveness:** Designing a user-friendly and visually appealing user interface that is responsive across different devices and screen sizes proved to be a challenging aspect, especially when dealing with complex layouts and dynamic content.
- **Error Handling and Debugging:** Implementing robust error handling and logging mechanisms to catch and handle unexpected errors, as well as debugging issues during development, required significant effort.