Advanced Web Programming Course Project Report

Tasks Management System (SPA)

Done By:

Rand Nofal.

Shams Abd Al-Aziz.

Under the guidance of:

Dr. Amjad Abu Hassan

Spring 2025

1. Introduction:

The **Tasks Management System** is a web-based application designed to help students and administrators stay organized, manage tasks, and communicate more easily. It works as a **Single Page Application (SPA)**, which means users can move around the app quickly without waiting for pages to reload, this means only the content changes.

This system makes it easier for students to keep track of their assignments and tasks, while giving administrators tools to assign tasks, follow progress, and chat directly with students. The app is built to be user-friendly and includes helpful features like a dark theme and real-time date and time display.

2. Project Requirements:

A. Functional Requirements:

- **User Authentication**
  The application allows users to create an account (sign up) and securely log in (sign in). A "Stay signed in" option is implemented.

- **User Roles**
  the system supports two user roles: **Students** and **Administrators**. Role-based access control is used to determine what each user can see and do within the application.

- **Project Management**
  Administrators can create projects with a title, description, deadline, and status, and assign them to specific students. Each student can log in and view only the projects assigned to them directly from their dashboard.

- **Task Management**
  Users can create and view tasks from their dashboard. Tasks can be assigned to specific users, and administrators can track task progress.

- **Student Information**
  Students can input their **university ID** during profile setup or editing. This ID helps administrators identify and assign tasks to students efficiently.

- **Chat System**
  a built-in chat feature allows real-time or near real-time messaging between administrators and students.

- **Date and Time Display**
  the application displays the current date and time within the UI. This helps users stay mindful of deadlines, meeting schedules, and upcoming tasks.

- **Dark Theme**
  a dark mode theme is applied throughout the application for better aesthetics and improved usability in low-light environments.

- **Single Page Application (SPA)**
  The system is designed as a SPA using front-end technologies that prevent full page reloads. This ensures smooth navigation, faster interactions, and an app-like user experience.

B. Non-Functional Requirements

- **Usability**
  The application features a clean and intuitive layout, making it easy for users to navigate and interact with its features without the need for technical assistance.

- **Performance**
  Optimizations were made to ensure fast loading times and fluid page transitions. The SPA structure minimizes reloads and ensures a responsive feel during use.

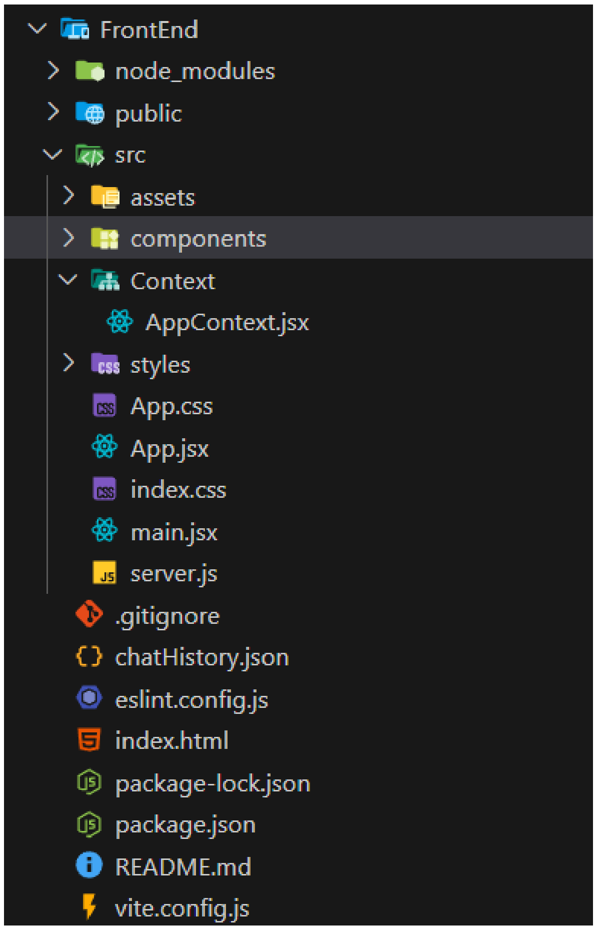
- **Scalability**
  the codebase follows a modular, component-based structure that allows for easy expansion. New features and users can be added in the future without disrupting the current system.

- **Authorization**
  Role-based authorization is enforced throughout the system. Students and administrators have access only to the features and data relevant to their roles, ensuring proper access control and data security.

3. Project Structure:

A. Front-End Structure:

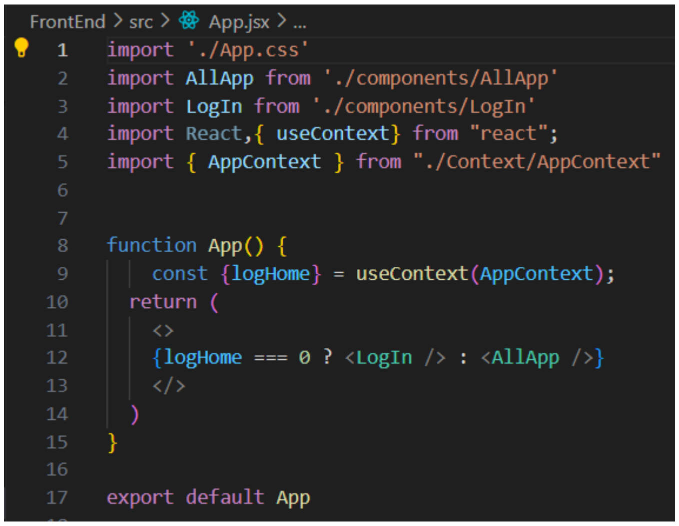
The front-end of the application was developed in two main phases. In the first phase, we built the interface using basic web technologies—HTML, CSS (with LESS), and JavaScript. This helped us focus on getting the layout and responsiveness right, while using local storage on browser to simulate how tasks and user data would work in a real environment. Once the basic version was working smoothly, we moved on to the second phase, where we refactored the entire front-end using React which is easier because it based on components. We also integrated Tailwind CSS to simplify styling and ensure a consistent, modern look across all pages. This transition allowed us to build a more dynamic, scalable, and maintainable application using reusable components and proper state management.



```
FrontEnd
  node_modules
  public
  src
    assets
    components
    Context
      AppContext.jsx
    styles
    App.css
    App.jsx
    index.css
    main.jsx
    server.js
  .gitignore
  chatHistory.json
  eslint.config.js
  index.html
  package-lock.json
  package.json
  README.md
  vite.config.js
```

As the picture above shows, when the project's Front End converted to React using **react dom library**, we make components it is jsx files, which contains code html with js, and we assign a css code to style it, and we can use it wherever we want, and this is the big advantage of React which is the reusability of the components. Also, we made a AppContext file to make a global state, so we can change the content of the page, without loading anything. We set the dev script in package.json to vite, to run the react project correctly.

And this is an example of component file, which we import 2 components in it, login and home components, then we use states to set the correct component to appear to the user, so we can use the App compenent wherever we want.



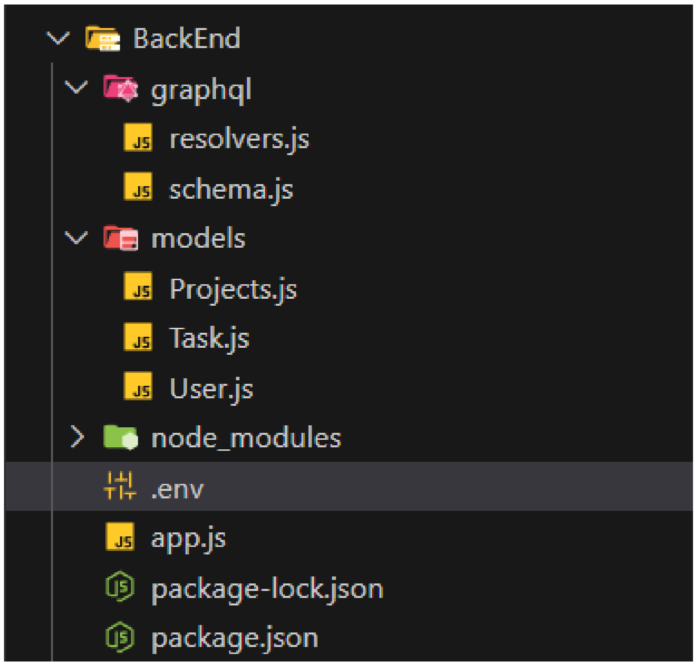
```
import './App.css'
import AllApp from './components/AllApp'
import LogIn from './components/LogIn'
import React,{ useContext} from "react";
import { AppContext } from "./Context/AppContext"


function App() {
    const {logHome} = useContext(AppContext);
  return (
    <>
    {logHome === 0 ? <LogIn /> : <AllApp />}
    </>
  )
}

export default App
```

B. Back-End Structure:

The back end of the application was built using Node.js to manage server-side routing and logic. We used **GraphQL** as the API layer to handle structured and efficient communication between the front-end and the database. For data storage, we chose **MongoDB** because it offers flexibility in handling complex and nested data structures—ideal for managing users, tasks and projects, we choose **Apollo Server** to handle the requests.

The back-end is responsible for handling core functionalities like authentication (login, signup), authorization (role-based access for admins and students), and all **CRUD** operations (Create, Read, Update, Delete) related to users, tasks, projects, and chat messages.

Overall, the structure was designed to keep the code clean and modular, making it easier to scale or add new features in the future.



```
BackEnd
  graphql
    resolvers.js
    schema.js
  models
    Projects.js
    Task.js
    User.js
  node_modules
  .env
  app.js
  package-lock.json
  package.json
```

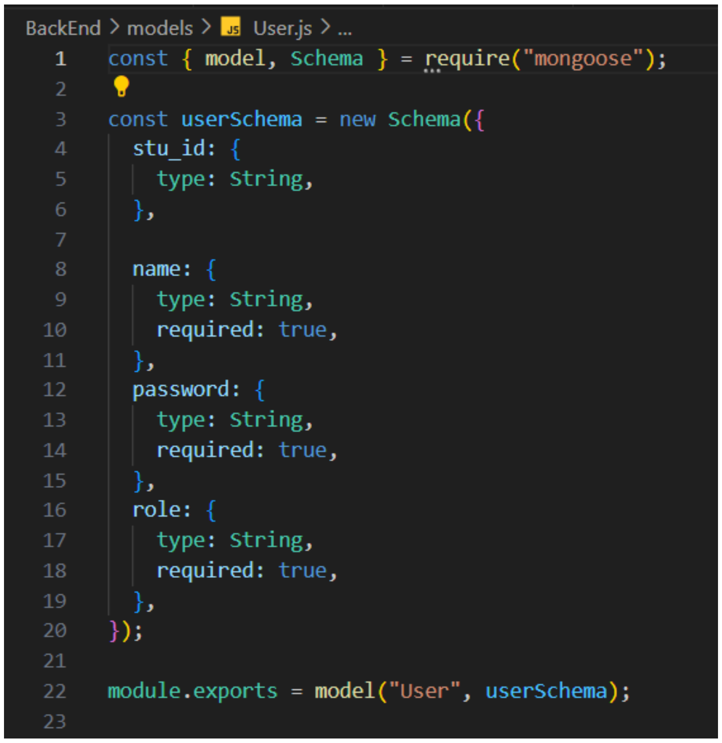
so, at the first we add the APIs, so we create the graphql folder which contains the APIs which handle the request and response, this folder contains schema of the data in database, and describe how we want to the data to be, then create the resolvers which handle the requests from server side (the requests that come from client side), so it take the request and analyze it then go to the DB to deal with data, then return the response.

Then we made the models folder, which is necessary for the MongoDB, so we want to make 3 models (like tables), task, user and project, then we put the data fields names and types for each model, depends on how we want to design our DB.

So, this is small explain of the work.

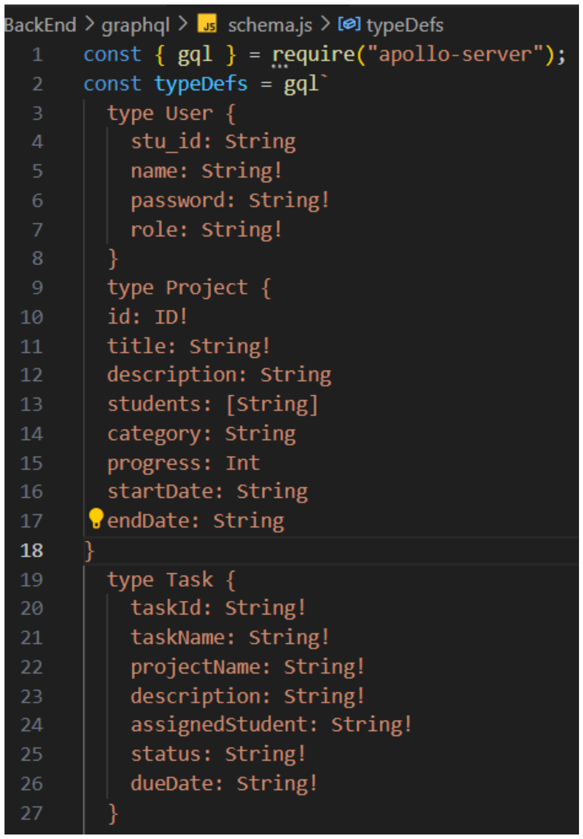
This is User model, how we want to user to be in DB, so we made a schema using **mongoose library**, so we put the data that we want to assign for each user, and the type of the data, then put if it is required for each user or not. Here for example each user could have student id of type string and must have name and password and role of type string.

Then we export this to use it in resolvers, so we reach the data easily from DB. And do the same for task and project.



BackEnd > models > User.js > ...

```
const { model, Schema } = require("mongoose");

const userSchema = new Schema({
  stu_id: {
    type: String,
  },

  name: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  role: {
    type: String,
    required: true,
  },
});

module.exports = model("User", userSchema);
```



BackEnd > graphql > schema.js > typeDefs

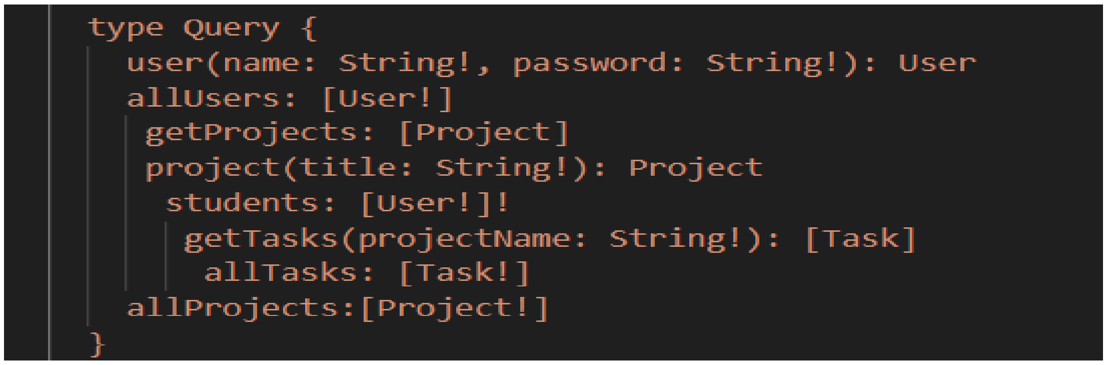
```
const { gql } = require("apollo-server");
const typeDefs = gql`
  type User {
    stu_id: String
    name: String!
    password: String!
    role: String!
  }
  type Project {
  id: ID!
  title: String!
  description: String
  students: [String]
  category: String
  progress: Int
  startDate: String
  endDate: String
}
  type Task {
    taskId: String!
    taskName: String!
    projectName: String!
    description: String!
    assignedStudent: String!
    status: String!
    dueDate: String!
  }
```

Then we made the schema.js to set the requests that the client can send and set the schema of the data, to accept and handle the requests, and set the type of response for each request type.

So, at the first we define the 3 types, which is the same in DB, and the data fields name and type form each model.

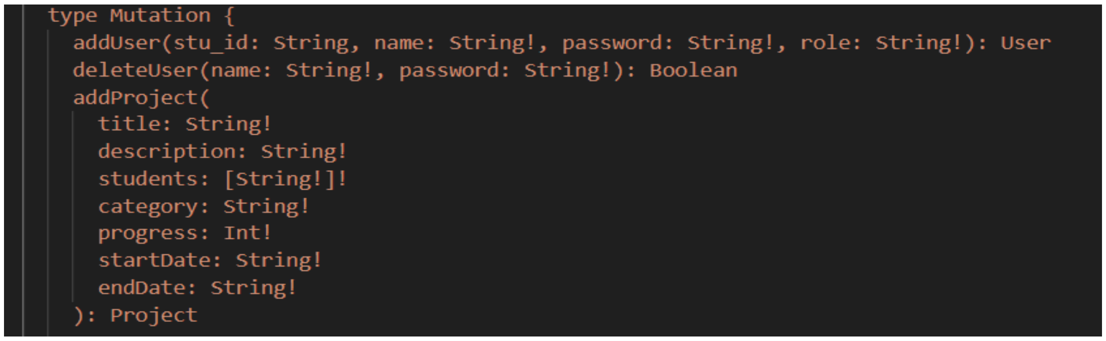
For task, each one must have id, name. project name, description, assigned student, status and due date.

Then we write the Queries, and define what the GET requests that the client can do, to handle it in the resolvers from DB.



```
type Query {
  user(name: String!, password: String!): User
  allUsers: [User!]
   getProjects: [Project]
   project(title: String!): Project
    students: [User!]!
     getTasks(projectName: String!): [Task]
      allTasks: [Task!]
  allProjects:[Project!]
}
```

After that write the Mutation, and define what the PUT, DELETE, POST requests that the client can do, to handle it in the resolvers from DB.



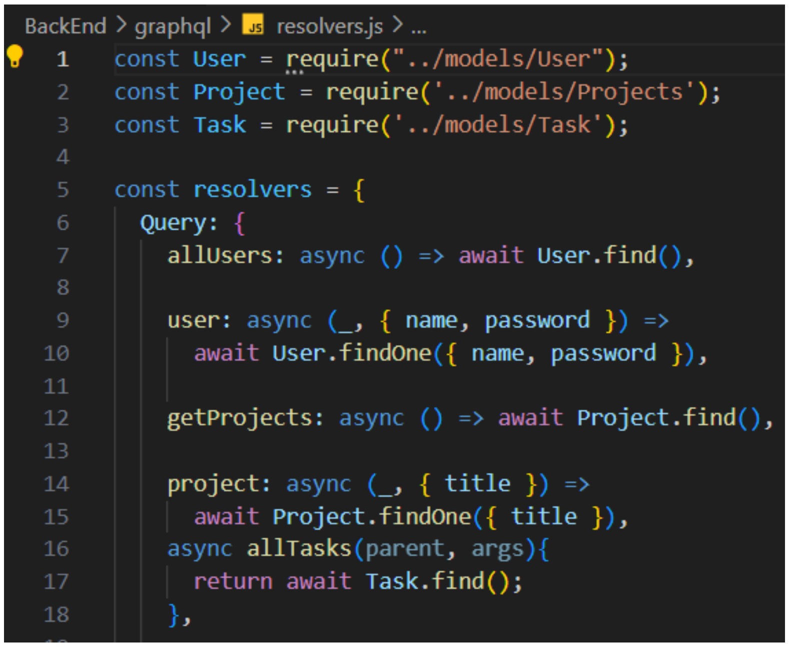
```
type Mutation {
  addUser(stu_id: String, name: String!, password: String!, role: String!): User
  deleteUser(name: String!, password: String!): Boolean
  addProject(
    title: String!
    description: String!
    students: [String!]!
    category: String!
    progress: Int!
    startDate: String!
    endDate: String!
  ): Project
```

So, here for example the client can make a request to add User, the client should enter the new user's data, and the result will be object of user type, and it will be added to DB, or any data storage. Also, it can make a query to return all users, the server will return a list of users of type user, or he can return specific user's data depends on its name and password, and so on.

For resolvers, we import the three models of our DB, then handle all the requests that we put on the schema file, which is possible requests from client.

So, we handle all Queries which is GET requests, so if the user wants all users, then the resolver from server side go to DB and get all users, it will return array of user's objects.

And we use async and await to make the function asynchronous, so there's no blocking for other tasks and wait for the response to come. Also, in some requests the users send arguments to get specific data with some limitations, like get user with some name and password, to check if it is real user to login.

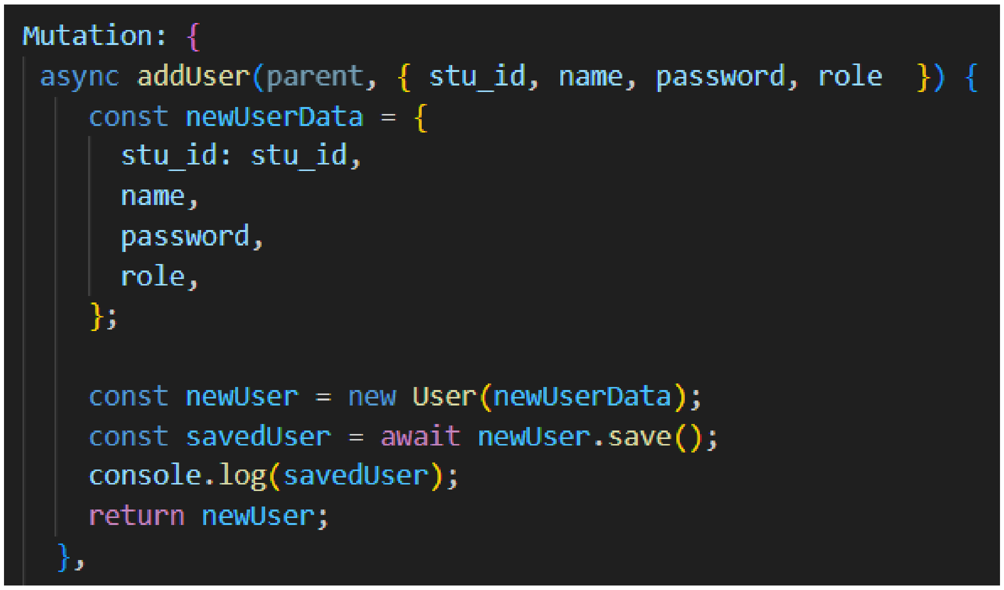


BackEnd > graphql > resolvers.js > ...

```
const User = require("../models/User");
const Project = require('../models/Projects');
const Task = require('../models/Task');

const resolvers = {
  Query: {
    allUsers: async () => await User.find(),

    user: async (_, { name, password }) =>
      await User.findOne({ name, password }),

    getProjects: async () => await Project.find(),

    project: async (_, { title }) =>
      await Project.findOne({ title }),
    async allTasks(parent, args){
      return await Task.find();
    },
```

Then we handle the Mutations, which is meaning handle the requests with method PUT, DELETE, POST.

Like this request to add user to DB, so when the client wants to add user, the request must have some info, then the request will handle from server side like this, make a new object of user and save it on DB, then return a response of the client side with object of new user that added successfully.

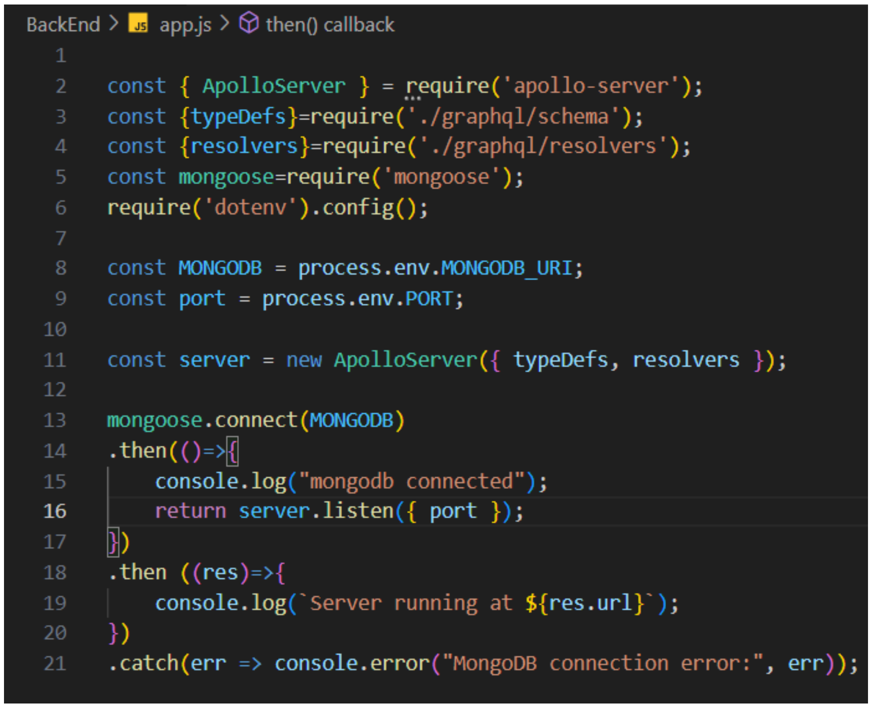


```
Mutation: {
 async addUser(parent, { stu_id, name, password, role  }) {
   const newUserData = {
     stu_id: stu_id,
     name,
     password,
     role,
   };

   const newUser = new User(newUserData);
   const savedUser = await newUser.save();
   console.log(savedUser);
   return newUser;
 },
```

after setup the DB and APIs, we set the app.js, which is the file to open the server and set it on some port. So, we use the Apollo server library, and mongoose for DB, and we import schema and resolvers to set them with DB.

so, we connect the DB with some URI, then let server listen for any requests that come from client side to server side to handle it from DB that connected with.

Also, we use **.env**, which is file to store secure data, like DB username and password and the port the server will listen on for any coming requests.

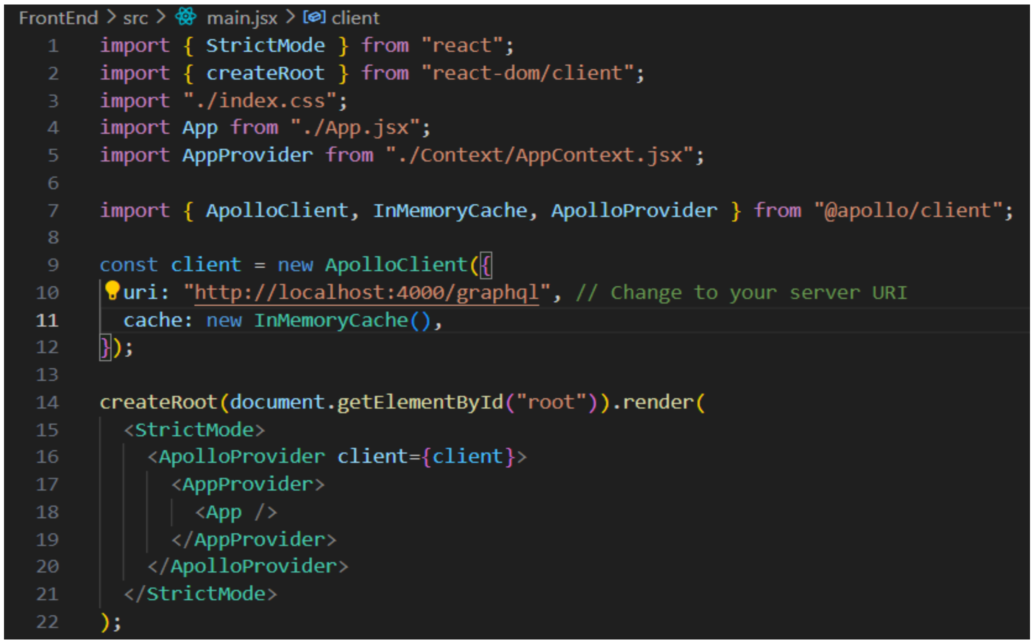


BackEnd > app.js > then() callback

```
const { ApolloServer } = require('apollo-server');
const {typeDefs}=require('./graphql/schema');
const {resolvers}=require('./graphql/resolvers');
const mongoose=require('mongoose');
require('dotenv').config();

const MONGODB = process.env.MONGODB_URI;
const port = process.env.PORT;

const server = new ApolloServer({ typeDefs, resolvers });

mongoose.connect(MONGODB)
.then(()=>{
    console.log("mongodb connected");
    return server.listen({ port });
})
.then ((res)=>{
    console.log(`Server running at ${res.url}`);
})
.catch(err => console.error("MongoDB connection error:", err));
```

Now we set the backend environment, to connect the BE with FE, in main.jsx file we do:

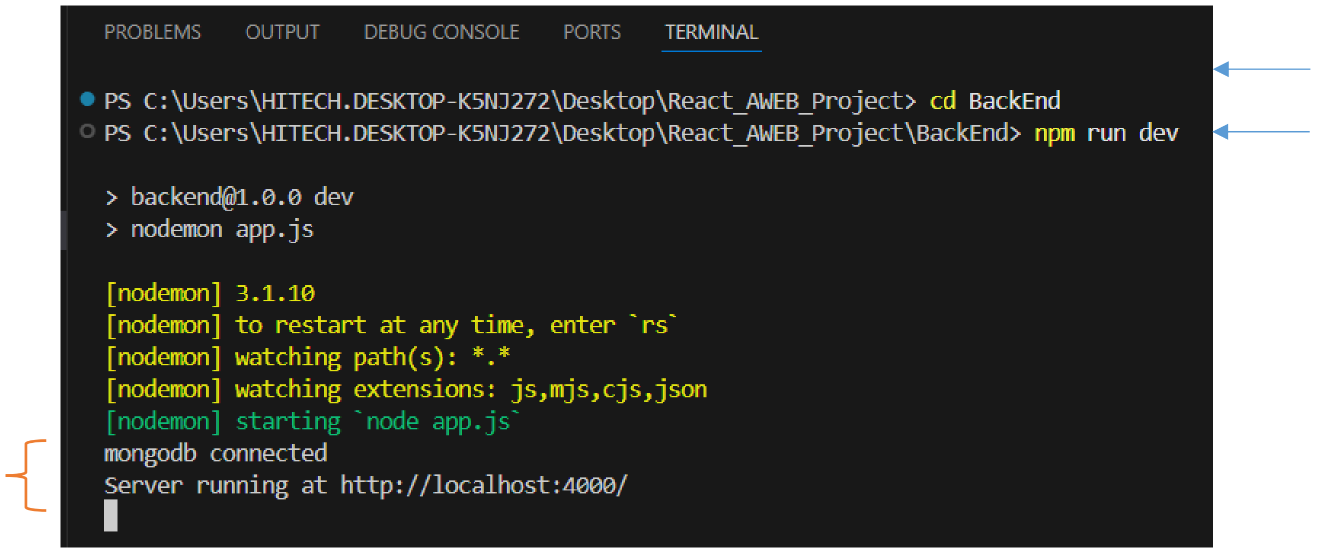
Import apollo client library, to make the requests to send to Apollo server, and make an object of it, then connect it with server URL, then render the components on the root page.



FrontEnd > src > main.jsx > client

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./index.css";
import App from "./App.jsx";
import AppProvider from "./Context/AppContext.jsx";

import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";

const client = new ApolloClient({
  uri: "http://localhost:4000/graphql", // Change to your server URI
  cache: new InMemoryCache(),
});

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <ApolloProvider client={client}>
      <AppProvider>
        <App />
      </AppProvider>
    </ApolloProvider>
  </StrictMode>
);
```

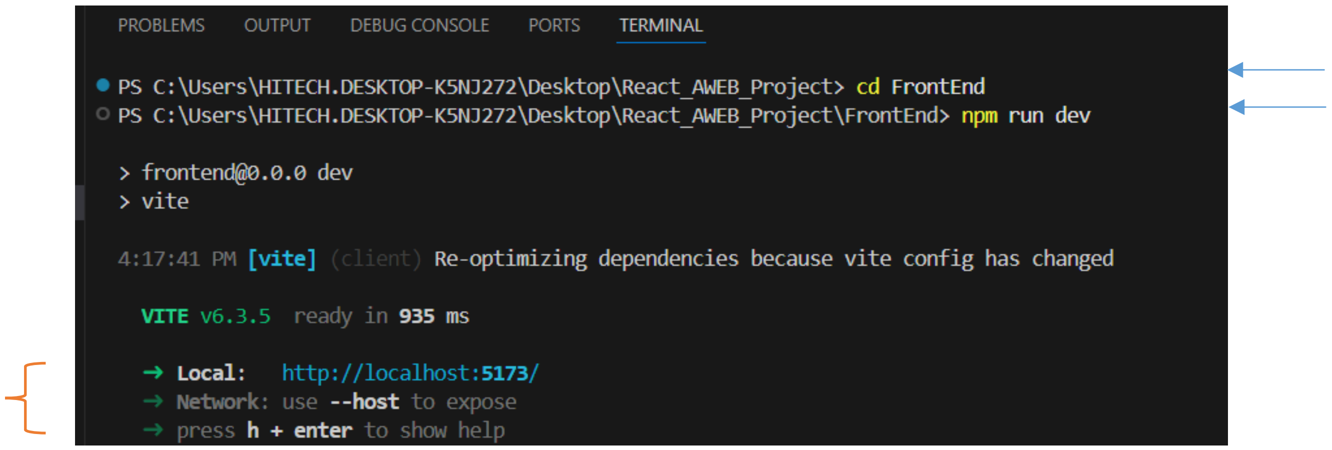
To run the server:



PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL

```
PS C:\Users\HITECH.DESKTOP-K5NJ272\Desktop\React_AWEB_Project> cd BackEnd
PS C:\Users\HITECH.DESKTOP-K5NJ272\Desktop\React_AWEB_Project\BackEnd> npm run dev

> backend@1.0.0 dev
> nodemon app.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
mongodb connected
Server running at http://localhost:4000/
```

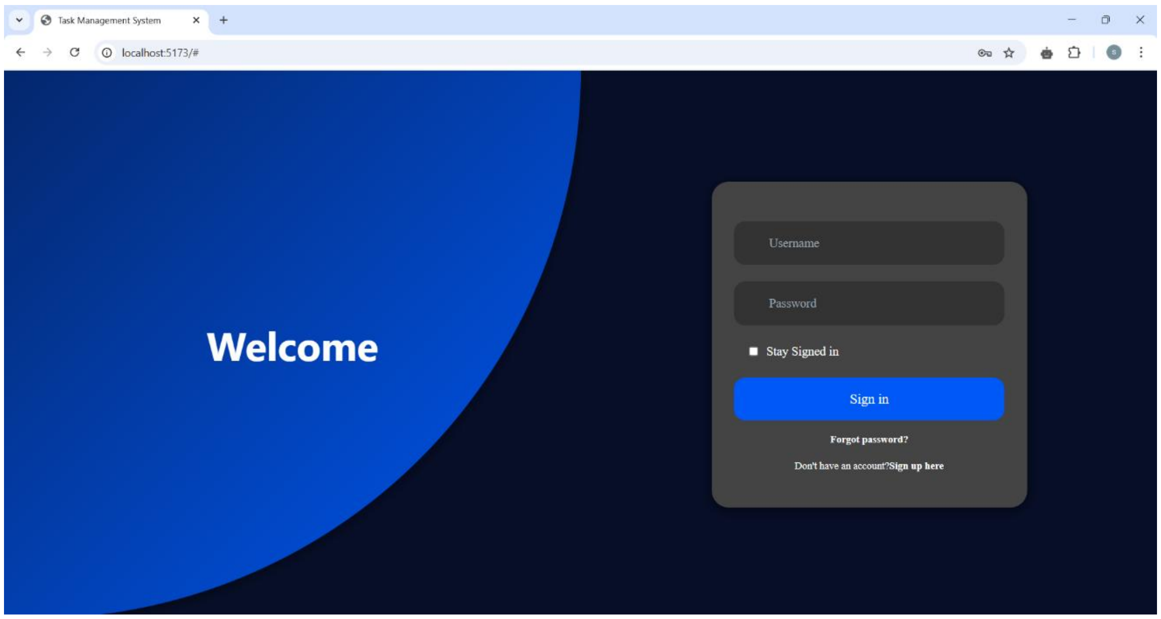
To run the Front End, while the BE running:



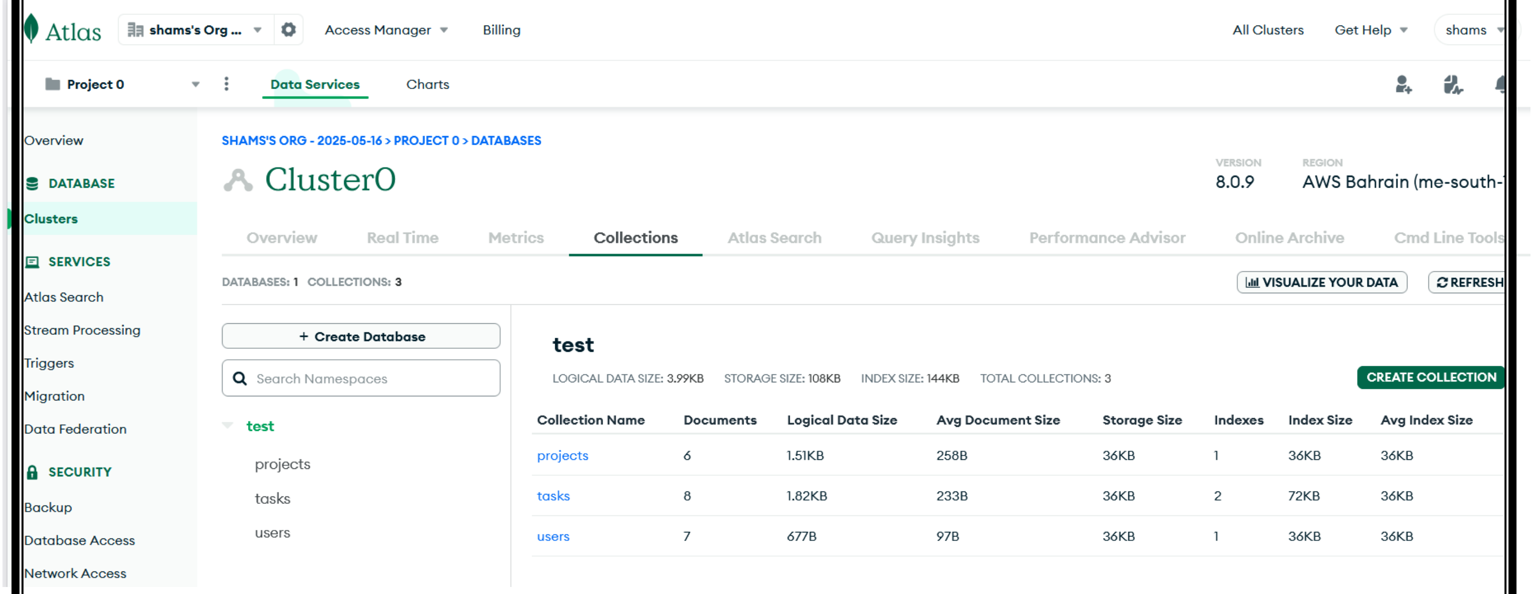
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL

```
PS C:\Users\HITECH.DESKTOP-K5NJ272\Desktop\React_AWEB_Project> cd FrontEnd
PS C:\Users\HITECH.DESKTOP-K5NJ272\Desktop\React_AWEB_Project\FrontEnd> npm run dev

> frontend@0.0.0 dev
> vite

4:17:41 PM [vite] (client) Re-optimizing dependencies because vite config has changed

  VITE v6.3.5  ready in 935 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

then go to http://localhost:5173/ to run application and get and return data to and from DB, using FE.



and this is the DB:



SHAMS'S ORG - 2025-05-16 > PROJECT 0 > DATABASES

Cluster0

VERSION 8.0.9 REGION AWS Bahrain (me-south-

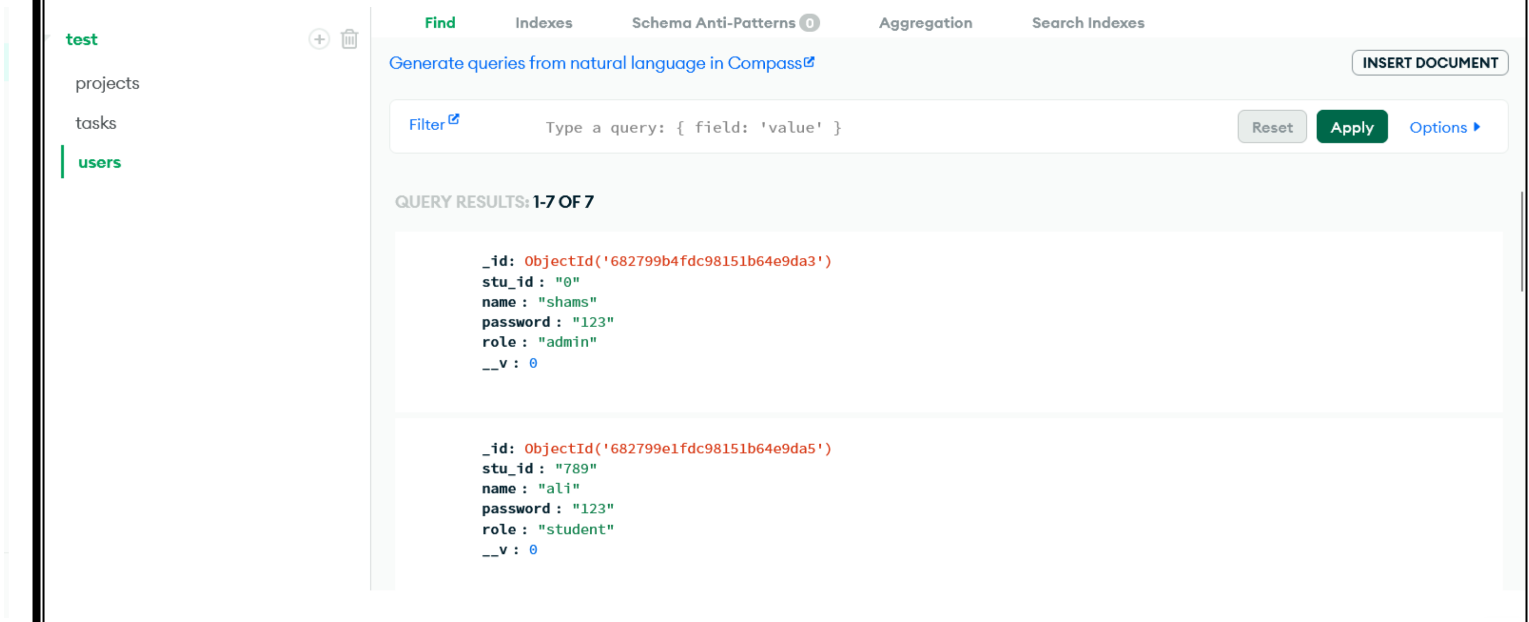
DATABASES: 1 COLLECTIONS: 3

test

LOGICAL DATA SIZE: 3.99KB STORAGE SIZE: 108KB INDEX SIZE: 144KB TOTAL COLLECTIONS: 3

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
projects	6	1.51KB	258B	36KB	1	36KB	36KB
tasks	8	1.82KB	233B	36KB	2	72KB	36KB
users	7	677B	97B	36KB	1	36KB	36KB

we have a 3 collections and models or tables.



QUERY RESULTS: 1-7 OF 7

```
_id: ObjectId('682799b4fdc98151b64e9da3')
stu_id : "0"
name : "shams"
password : "123"
role : "admin"
__v : 0

_id: ObjectId('682799e1fdc98151b64e9da5')
stu_id : "789"
name : "ali"
password : "123"
role : "student"
__v : 0
```

when we press on users model or table, the list of users appears, and each user as object, with its info.

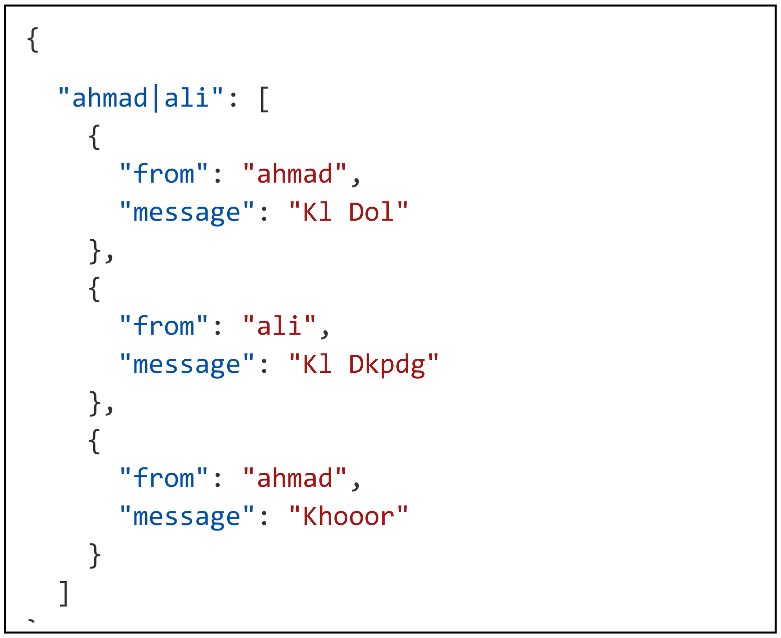
4. Real-Time Chat System

The application features a real-time chat system that allows administrators and students to communicate instantly. This system is powered by WebSocket, enabling live messaging without the need to refresh the page. When a user sends a message, it is transmitted over an active WebSocket connection and delivered instantly to the intended recipient, ensuring fast and responsive communication.

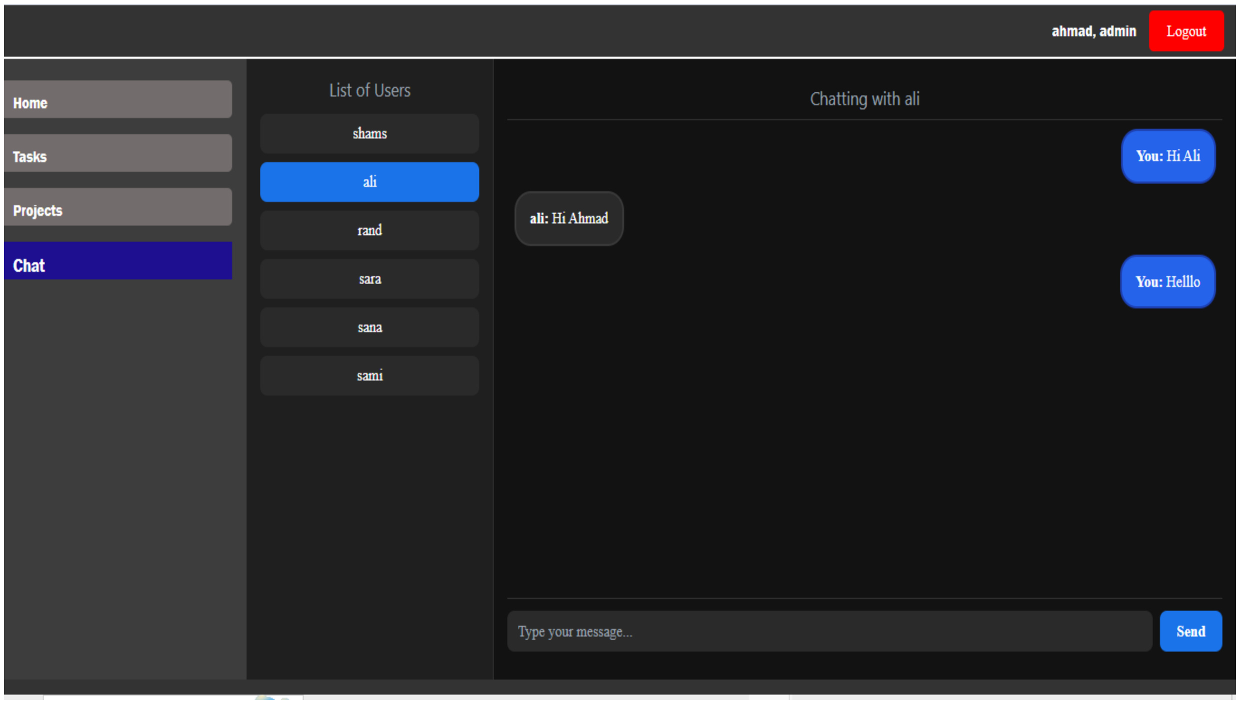
To ensure data privacy and security, all chat messages are encrypted before being saved. Instead of storing messages in a traditional database, we use a JSON file to keep a lightweight, file-based message log. Each message is encoded using a simple encryption method (Caesar cipher), making the data unreadable without decryption.

When users open the chat, the system reads the encrypted messages from the file, decrypts them on the fly, and displays them in the chat interface.

This setup provides a simple and secure way to manage real-time messaging, while keeping the system easy to manage during early stages of development.



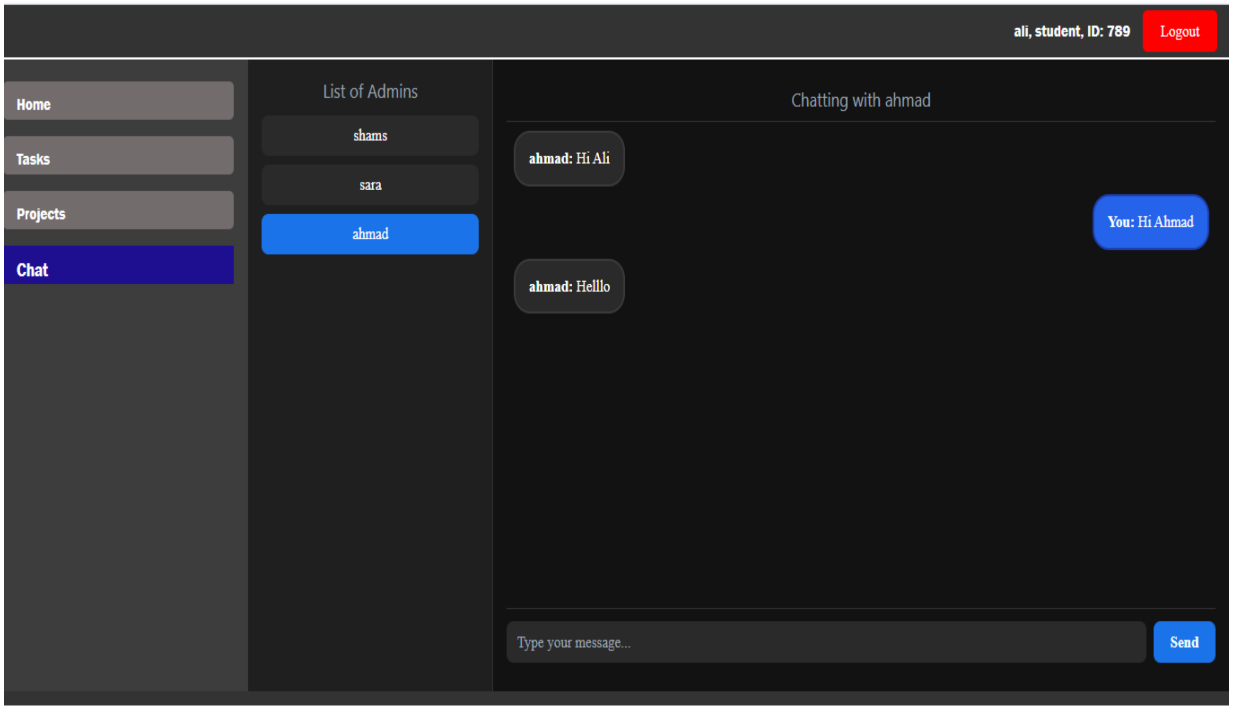
```
{
  "ahmad|ali": [
    {
      "from": "ahmad",
      "message": "Kl Dol"
    },
    {
      "from": "ali",
      "message": "Kl Dkpdg"
    },
    {
      "from": "ahmad",
      "message": "Khooor"
    }
  ]
```



You: Hi Ali

ali: Hi Ahmad

You: Helllo



ahmad: Hi Ali

You: Hi Ahmad

ahmad: Helllo

Here is the Demo link for our application with all features with FE and BE:

https://drive.google.com/file/d/19kKPhqNjZ8eb06Syg3CKaHnf9Gey3ibw/view?usp=drivesdk

and here is the link of GitHub repository:

https://github.com/ShamsAziz03/Task-Management-System.git

Conclusion

The Tasks Management System successfully integrates a full-stack web application using modern technologies like React, Node.js, GraphQL, and MongoDB. It delivers a responsive, user-friendly SPA experience for students and administrators to manage tasks, projects, and real-time communication efficiently. The modular structure, role-based access, and clean codebase ensure scalability, security, and ease of future development.