FREELANCING APPLICATION MERN

A PROJECT REPORT

SUBMITTED BY

MEGANA.S - 310821104055

DHANALAKSHMI.B - 310821104024

ANUSUYA.B - 310821104011

UMA MAHESHWARI.S – 310821104104

IN PARTIAL FULFILLMENT FOR THE AWARD OF THE DEGREE OF BACHELOR OF ENGINEERING IN

COMPUTER SCIENCE ENGINEERING

JEPPIAAR ENGINEERING COLLEGE

ANNA UNIVERSITY

CHENNAI – 600025

NOVEMBER 2024

ANNA UNIVERSITY:CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report “FREELANCING APPLICATION MERN” is the

bonafide work of

“MEGANA.S(310821104055)”,“DHANALAKSHMI.B(310821104024)”,“ANUSUYA.B(310821104011)”, “UMA MAHESHWARI.S(310821104104)” who

carried out the project under supervision.

SUPERVISOR	HEAD OF THE DEPARTMENT
JEEVITHA.D	DR.J.ANITHA GNANSELVI

TABLE OF CONTENTS

INTRODUCTION :

This report outlines the development of a Freelancing Application project, created using the MERN (MongoDB, Express.js, React.js, Node.js) stack. The project was developed collaboratively by our team, consisting of Meghana S., Dhanalakshmi B., Uma Maheshwari S., and Anusuya B., with well-defined roles and contributions.

The scope of the project and the respective team member responsibilities are as follows:

1. Frontend Development:
   Megana S. took the lead in designing and implementing a user-friendly interface using React.js.
2. Backend Development:
   Dhanalakshmi B. was responsible for building a robust server-side architecture with Node.js and Express.js, integrated with MongoDB for data management.
3. Demo Video, Files and Packages :
   Uma Maheshwari S. created a comprehensive video walkthrough of the

application, showcasing its features and functionality, along with backend implementation links. Ensured the preparation of all essential project files and packages-source code.

4. Report Preparation:

   Anusuya B. prepared this detailed report, documenting the development process, challenges faced, solutions implemented, and project outcomes.

This freelancing application aims to provide a seamless platform for connecting freelancers with potential clients, focusing on features like user authentication, project listings, and secure communication. This report elaborates on our methodology, tools, and technologies used, and provides insights into our collaborative efforts and results.

Project Overview :

Purpose

The Freelancing Application is designed to provide an intuitive and efficient platform for freelancers and clients to connect, collaborate, and complete projects. With the rising demand for freelance work, this application addresses the need for a secure, streamlined environment where freelancers can showcase their skills and clients can find suitable talent for their projects.

The primary objectives of the project are:

- To simplify the process of hiring freelancers by offering a user-friendly interface.
- To ensure secure and reliable communication between freelancers and clients.
- To enable efficient project management through seamless features for posting, browsing, and completing job opportunities.

Features

The application includes the following core features:

1. User Authentication:

   - Secure sign-up and login for both freelancers and clients.

   - Role-based access to features depending on the user type.

2. Profile Management:

   - Freelancers can create and manage profiles with skillsets, portfolios, and experience details.

- Clients can manage profiles to reflect their project requirements and preferences.

3. Job Posting and Search:

- Clients can post detailed job opportunities, specifying required skills and budgets.

- Freelancers can search and filter jobs based on skills, categories, and budgets.

4. Proposal System:

- Freelancers can submit proposals for jobs, allowing clients to review and shortlist candidates.

5. Project Management:

- A dashboard for tracking the status of active projects.

- Communication tools to facilitate discussions between freelancers and clients.

6. Payment Integration:

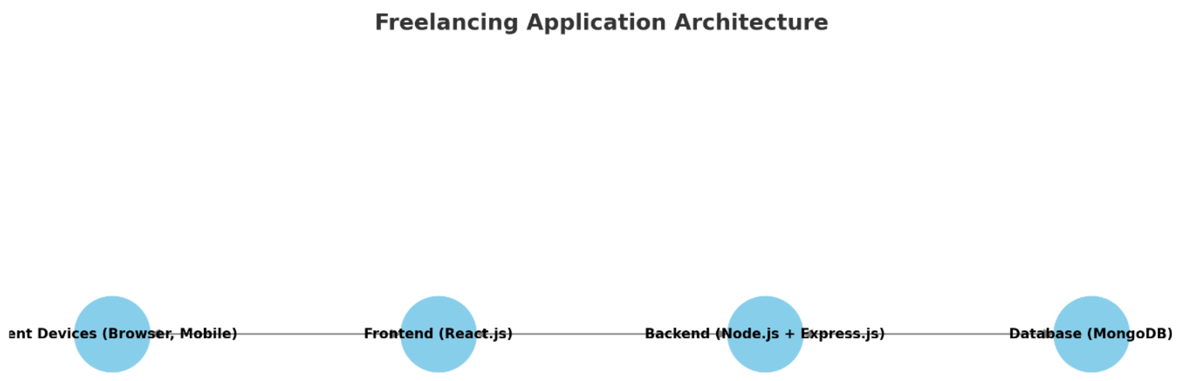
- Secure payment gateways to ensure safe transactions between clients and freelancers.

7. Admin Pane

- Admin-level controls to manage user accounts, handle disputes, and monitor system health.

This application leverages the MERN stack to deliver a scalable and efficient solution tailored to the needs of the freelancing community.

Architecture :


Freelancing Application Architecture
ent Devices (Browser, Mobile)
Frontend (React.js)
Backend (Node.js + Express.js)
Database (MongoDB)

Here is the architecture diagram for the Freelancing Application. It illustrates the interactions between the key components:

1. Client Devices (Browser, Mobile): End-user devices interacting with the application.
2. Frontend (React.js): The user interface handling input and displaying data.

3. Backend (Node.js + Express.js): The server-side logic managing requests and processing data.

4. Database (MongoDB): The data storage system for users, projects, and other application information.

he Architecture section with details about the
    frontend, backend, and database:

The Freelancing Applicationis built using the MERN (MongoDB, Express.js, React.js, Node.js) stack. This modern web development architecture ensures scalability, responsiveness, and maintainability, making it well-suited for a robust application.

1. Frontend

The frontend of the application is developed using React.js, a popular JavaScript library for building interactive user interfaces.

- Technologies Used: React.js, HTML5, CSS3, JavaScript, and Material-UIBootstrap for UI components.

- Key Features:

- Dynamic and responsive UI for a seamless user experience across devices.

-Reusable components for efficiency and scalability.

-API integration with the backend to display real-time data.

-Role-based navigation and user interaction tailored for freelancers and clients.

2. Backend

The backend is built using Node.js with Express.js, providing a robust and scalable server-side framework for handling business logic and managing API routes.

- Technologies Used: Node.js, Express.js.
- Key Features:
  - RESTful APIs for communication between the frontend and backend.
  - Middleware for handling user authentication (e.g., JWT) and request validation.
  - Scalability to manage high volumes of concurrent users and data transactions.

- Error handling and logging to ensure smooth operation.

3. Database

The application uses MongoDB, a NoSQL database, for data storage and management. MongoDB is chosen for its flexibility in handling unstructured and semi-structured data, which aligns with the dynamic nature of freelancing projects.

- Technologies Used: MongoDB, Mongoose (for object data modeling).
- Key Features:
  - User-friendly schema design for managing data like user profiles, job postings, and project details.
  - High performance with support for fast read and write operations.
  - Scalability to handle growing data and user demands.
  - Secure storage with appropriate data encryption measures.

Integration

The frontend communicates with the backend via RESTful APIs, exchanging JSON data. The backend interacts with the MongoDB database using Mongoose for seamless data retrieval, updates, and storage. This architecture ensures a modular and scalable solution that supports the application's core functionalities efficiently.

Setup Instructions :

Prerequisites

Before setting up the application, ensure you have the following tools and software installed on your system:

1. Node.js and npm:

   - Download and install Node.js from the [official website](https:nodejs.org).

   - Verify installation:

```
node -v
npm -v
```

2. MongoDB:

- Install MongoDB locally or use a cloud-based service like [MongoDB Atlas](https:www.mongodb.comcloudatlas).

   - Start MongoDB server if running locally.

3. Git:

   - Install Git from the [official website](https:git-scm.com).

   - Verify installation:

```
git --version
```

4. Code Editor:

   - Recommended: [Visual Studio Code (VS Code)](https:code.visualstudio.com).

5. Browser:

   - A modern browser like Google Chrome or Firefox for testing the application.

Installations

Follow these steps to set up the application locally:

1. Clone the Repository:

   Open a terminal and run:

```
git clone <repository-url>
cd <repository-folder>
```

2. Install Dependencies:

Navigate to the frontend and backend directories separately and install the required dependencies using npm.

For the backend:

```
cd backend
npm install
```

For the frontend:

```
cd frontend
npm install
```

3. Set Up Environment Variables:

- Create a `.env` file in the backend directory.
- Add the following configuration (adjust values as per your setup):

```
.env
PORT=5000
MONGO_URI=<your-mongodb-connection-string>
JWT_SECRET=<your-jwt-secret>
```

- For MongoDB, if using MongoDB Atlas, replace `<your-mongodb-connection-string>` with the connection string from your Atlas cluster.

4. Run the Application:

   Start the backend server:

   npm start

   Start the frontend development server:

   cd ..frontend

   npm start

5. Access the Application:

   - Open your browser and navigate to:

     http:localhost:3000

   - The frontend will interact with the backend running on port `5000` (default setup).

6. Testing:

   - Verify all features are working as expected, including user authentication, job posting, and project management.

For a MERN (MongoDB, Express.js, React, Node.js) application like the freelancing application you're working on, a clean and modular folder structure is essential for scalability, maintainability, and clarity.

Below is a detailed folder structure with an explanation of each directory and file:

Folder Structure:

```
freelancing-app
├── client                React frontend
|   ├── public             Public assets
|   |   ├── index.html       Main HTML file
|   ├── src               All React app source code
|   |   ├── components       Reusable components
(Buttons, Modals, etc.)
|   |   ├── context        React Context API for
global state management
|   |   ├── pages          Pages for different routes
(Home, Login, etc.)
|   |   ├── services       API calls and interaction
with backend
|   |   ├── styles         Global and component-
specific styles (CSSSASS)
|   |   ├── App.js          Main app component
|   |   └── index.js        Entry point to React app
├── server                Node.js backend
|   ├── config            Configuration files (DB,
auth, etc.)
|   |   ├── db.js          MongoDB connection setup
|   |   └── config.js       Other configuration
settings (e.g., JWT secrets)
|   ├── middleware         Custom middleware
(authentication, validation, etc.)
|   |   └── authMiddleware.js
|   ├── models             MongoDB models
(schemas)
|   |   ├── User.js         User schema
|   |   └── Job.js         Job post schema
|   ├── routes            API route definitions
|   |   ├── authRoutes.js    Authentication routes
(Login, Register)
|   |   └── jobRoutes.js     Job-related routes
|   ├── utils             Utility functions and helpers
|   |   └── validation.js    Validation functions
```

| ├── .env   Environment variables for backend

| ├── server.js   Entry point to backend server

| └── package.json   Dependencies and scripts for backend

├── .gitignore   Git ignore file to avoid committing unwanted files

├── package.json   Root dependencies and scripts (for both client & server)

└── README.md   Project description and instructions

Detailed Explanation of Each DirectoryFile:

1. `client` (Frontend - React)

This folder contains all the code for the React frontend application.

- `public`:

- `index.html`: The main HTML file that serves as the template for the React app.

- `src`:

  - `assets`: This folder stores images, fonts, icons, or any other static assets you might use in your app.

  - `components`: This contains reusable React components like buttons, forms, modals, etc. You can break down your app into small, reusable pieces here.

  - `context`: If you are using the React Context API for global state management (e.g., user authentication status), this is where those files will reside.

  - `hooks`: Custom React hooks for handling logic that might be shared across different components.

  - `pages`: Each file in this folder represents a page in your app (e.g., Home.js, Login.js, Dashboard.js, etc.). These pages are usually routed by React Router.

  - `services`: This folder includes API service functions that interact with the backend. Here you can handle the logic for making requests to your Express API.

- `styles`: All styles related to your app. You can use CSSSASS to style your components.

- `App.js`: The root component of your app that initializes routing and includes global components like navigation bars, sidebars, etc.

- `index.js`: The entry point for the React app where ReactDOM renders the App component.

2. `server` (Backend - Node.js Express)

This folder contains all the code for your Node.js backend.

- `config`:

  - `db.js`: Contains logic to connect to MongoDB, typically using Mongoose.

  - `config.js`: Stores other configurations such as JWT secrets, third-party API keys, etc.

- `models`:

- `User.js`: MongoDB schema and model for user data.
- `Job.js`: MongoDB schema and model for job posts.

- `routes`:
  - `authRoutes.js`: Defines routes for login, registration, and authentication.
  - `jobRoutes.js`: Defines routes for job listings, creation, and applications.

- `utils`:
  - `validation.js`: Utility functions for validating inputs, such as email format validation or password strength checks.

- `server.js`: The entry point to your backend. This file will create an Express server, use middleware, and define routes.

- `.env`: Stores environment variables like database URL, JWT secret, etc., for the backend.

3. Root Files:

- `.gitignore`: Specifies files and directories that should not be tracked by Git (e.g., `node_modules`, `.env`, etc.).

- `package.json`: This file contains metadata about the project, such as dependencies, scripts for running the app (start, dev, etc.), and more.

- `README.md`: A readme file to explain the purpose of the project, setup instructions, and other relevant details for developers or users.

Running the App:

For a MERN stack app, you would typically run both the client and server in separate terminals:

1. Backend (NodeExpress):

   - Install dependencies: `npm install` in the `server` folder.

   - Run server: `npm start` or `node server.js`.

2. Frontend (React):

   - Install dependencies: `npm install` in the `client` folder.

   - Run React app: `npm start`.

This structure supports both development and scalability, keeping the client and server code organized and easy to maintain.

RUNNING THE APPLICATION :

To run the freelancing application built with the MERN stack (MongoDB, Express, React, Node.js), you need to follow a step-by-step process. Below are the detailed instructions for running both the client (React) and server (Node.js/Express) parts of the application.

Pre-requisites:

1. Node.js installed on your machine. If not, you can download and install it from [Node.js official website](https://nodejs.org/).
2. MongoDB installed locally or you can use MongoDB Atlas (a cloud database service) for a remote database.
3. Git (if you're using version control).
4. Your React frontend and Node.js backend project files must be in their respective folders (`client/` for frontend and `server/` for backend).

Steps to run the application:

1. Set Up the Backend (Server)

a. Install Backend Dependencies

1. Navigate to the `server/` directory using the terminal or command prompt:

```
cd path/to/freelancing-app/server
```

2. Install the required dependencies for the backend:

```
npm install
```

b. Configure Environment Variable

1. Create a `.env` file in the `server/` directory if it doesn't exist.

2. Add necessary environment variables in the `.env` file, such as:

- MongoDB connection string (`MONGO_URI`)
- JWT Secret (`JWT_SECRET`)
- Port for the server (`PORT`)

Example:

```
MONGO_URI=mongodb://localhost:27017/freelancing-app
  JWT_SECRET=your_jwt_secret
  PORT=5000
```

You can also use MongoDB Atlas for a cloud database connection by replacing `localhost` with your MongoDB Atlas URI.

c. Run the Backend Server

1. After installing dependencies and setting up the environment variables, you can run the backend server:

```
npm start
```

This will start the server, usually on port 5000 by default. You can open your browser and visit `http://localhost:5000` to check if the backend is running.

2. Set Up the Frontend (React Client)

a. Install Frontend Dependencies

1. Open a new terminal or command prompt and navigate to the `client/` directory:

```
cd path/to/freelancing-app/client
```

2. Install the required dependencies for the frontend:

```
npm install
```

b. Configure API Base URL

1. You need to ensure that the React frontend can communicate with the backend server.

2. In the `client/src/services/` folder (or wherever your API functions are stored), set the API base URL to the backend server URL (usually `http://localhost:5000`).

Example in `client/src/services/api.js`:

```javascript
const API_URL = "http://localhost:5000";  // Adjust if you're using a different port or cloud server

export const fetchJobs = async () => {
    const response = await fetch(`${API_URL}/jobs`);
    const data = await response.json();
    return data;
};
```

c. Run the Frontend Development Server

1. After installing the dependencies and setting up the API URL, you can start the React development server:

```
npm start
```

2. This will start the React development server, usually on port 3000. You can visit `http://localhost:3000` to check if the frontend is running.

3. Testing the Application

Once both the backend and frontend servers are running, you can:

- Visit the frontend at `http://localhost:3000` to interact with the application (viewing job listings, applying for jobs, etc.).
- Ensure that actions such as user authentication, job posting, and application are working as expected.
- Check if the frontend is successfully making API requests to the backend (check browser console/network tab for API calls).

By following these steps, you'll be able to run the freelancing application built with the MERN stack locally on your machine. Let me know if you need help with any specific part of the setup!.

API DOCUMENTATION :

Here’s a detailed API documentation structure for the Freelancing Application built with the MERN stack (MongoDB, Express, React, Node.js). This will include the routes for user authentication, job management, and any other key features in your application. The documentation will also describe how each API endpoint works, what data it expects, and the responses it returns.

Freelancing Application API Documentation

Base URL:

- For local development, the base URL will be: `http://localhost:5000`

- For production, replace with your deployed server's URL.

Authentication API

1. User Registration

- Endpoint: `POST /api/auth/register`
- Description: Registers a new user.
- Request Body:

```
json
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "password123",
  "role": "freelancer"  // or "client"
}
```

- Response:
  - Success (201):

```
json
{
  "message": "User registered successfully",
  "user": {
    "id": "userId",
    "name": "John Doe",
    "email": "john@example.com",
    "role": "freelancer"
    }
  }
- Error (400):
  json
  {
    "message": "Email already exists"
  }
```

2. User Login

- Endpoint: `POST /api/auth/login`
- Description: Authenticates the user and returns a JWT token.
- Request Body:

```
 json
 {
   "email": "john@example.com",
   "password": "password123"
 }
```

- Response:

```
 - Success (200):
   json
   {
     "message": "Login successful",
     "token": "jwt_token_here"
   }
```

- Error (401):

```json
   {
     "message": "Invalid credentials"
   }
```

3. Get User Profile

- Endpoint: `GET /api/auth/me`
- Description: Fetches the logged-in user's profile (requires JWT authentication).
- Headers:
  - Authorization: `Bearer <jwt_token>`
- Response:
  - Success (200):

```json
   {
     "user": {
      "id": "userId",
      "name": "John Doe",
      "email": "john@example.com",
      "role": "freelancer"
    }
  }
```

- Error (401):

```json
  {
    "message": "Not authorized"
  }
```

Job Management API

1. Create Job Post

- Endpoint: `POST /api/jobs`
- Description: Allows a client to create a job post.
- Request Body:

json

```
{
  "title": "Frontend Developer Needed",
    "description": "Looking for a frontend developer to
build a website.",
    "budget": 500,
    "clientId": "userId",  // Client's user ID
    "category": "Web Development"
  }
- Response:
  - Success (201):
    json
    {
      "message": "Job post created successfully",
      "job": {
        "id": "jobId",
        "title": "Frontend Developer Needed",
        "description": "Looking for a frontend developer to
build a website.",
        "budget": 500,
        "category": "Web Development",
        "clientId": "userId"
      }
    }
```

```

  - Error (400):

~~~
    json
    {
      "message": "Invalid data provided"
    }
~~~

2. Get All Job Posts

- Endpoint: `GET /api/jobs`

- Description: Fetches all available job posts.

- Response:

  - Success (200):

    ```json
    {
      "jobs": [
        {
          "id": "jobId1",
          "title": "Frontend Developer Needed",
          "description": "Looking for a frontend developer
to build a website.",
          "budget": 500,

```
        "category": "Web Development",
        "clientId": "userId1"
      },
      {
        "id": "jobId2",
        "title": "Backend Developer Needed",
        "description": "Looking for a backend developer
for an API project.",
        "budget": 600,
        "category": "Backend Development",
        "clientId": "userId2"
      }
    ]
  }
```

3. Get Job Post by ID

- Endpoint: `GET /api/jobs/:id`

- Description: Fetches a specific job post by its ID.

- Response:

  - Success (200):

    json

```
{
  "job": {
    "id": "jobId",
    "title": "Frontend Developer Needed",
    "description": "Looking for a frontend developer to build a website.",
    "budget": 500,
    "category": "Web Development",
    "clientId": "userId"
  }
}
```

- Error (404):

```
json
{
  "message": "Job post not found"
}
```

4. Update Job Post

- Endpoint: `PUT /api/jobs/:id`
- Description: Updates an existing job post (only by the client who created it).
- Request Body:

```
  json
  {
    "title": "Updated Job Title",
    "description": "Updated job description.",
    "budget": 700
  }
- Response:
  - Success (200):
    json
    {
      "message": "Job post updated successfully",
      "job": {
        "id": "jobId",
        "title": "Updated Job Title",
        "description": "Updated job description.",
        "budget": 700,
        "category": "Web Development",
        "clientId": "userId"
      }
    }
```

  - Error (400):

    json

    {

      "message": "Invalid data provided"

    }

5. Delete Job Post

- Endpoint: `DELETE /api/jobs/:id`

- Description: Deletes a job post (only by the client who created it).

- Response:

  - Success (200):

    json

    {

      "message": "Job post deleted successfully"

    }

  - Error (404):

    json

    {

      "message": "Job post not found"

    }

Job Application API

1. Apply for Job

- Endpoint: `POST /api/jobs/:id/apply`
- Description: Allows a freelancer to apply for a job.
- Request Body:

```
  json
  {
    "freelancerId": "userId",  // Freelancer's user ID
    "coverLetter": "I am very experienced in frontend development."
  }
```

- Response:
  - Success (201):

```
    json
    {
      "message": "Applied for the job successfully"
    }
```

  - Error (400):

```
    json
```

```
   {
     "message": "Already applied for this job"
   }
```

2. Get Applications for a Job

- Endpoint: `GET /api/jobs/:id/applications`
- Description: Fetches all applications for a specific job.
- Response:
  - Success (200):

```
   json
   {
     "applications": [
       {
         "freelancerId": "userId1",
         "coverLetter": "I am very experienced in frontend development."
       },
       {
         "freelancerId": "userId2",
         "coverLetter": "I am an expert in backend development."
```

```
      }
    ]
  }
```

This is the API documentation for the Freelancing Application built with MERN stack. It covers authentication (register, login, profile), job management (create, read, update, delete), and job applications (apply, view applications). Each API endpoint is designed to allow the frontend (React) to interact seamlessly with the backend (Node/Express) and MongoDB. Let me know if you need further clarification or additional endpoints!

AUTHENTICATION:

MERN stack-based freelancing application, authentication and authorization are essential to ensure that only authorized users can access specific resources and perform specific actions. Below, I’ll explain how these concepts are typically handled in such a system.

1. Authentication in MERN Application

"What is Authentication?"

Authentication refers to the process of verifying the identity of a user. in a freelancing application, this means ensuring that users (dients, freelancers, or admins) are who they say they are before granting them access to the application.

How Authentication Works:

1. "User Registration:"

-The user provides their details (eg, name, email, password, role).

-The server processes the registration request, validates the data, and stores the user's information in the database.

Passwords should always be hashed using a secure algorithm (eg, berypt) before storing in the database to prevent plaintext storage.

"Example:"

"name": "John Dee

"email": "John.doe@example.com",

"password": "securepassword123",

"rale": "freelancer"

2. "Login:"

-The user provides their email and password to log in

-The server checks if the email exists in the database if it does, & venfies the provided

password by comparing it to the hashed password stored in the database.

-If the credentials are vald, the server generates a "JWT USON Web Token)", which is sent back to the client

-The client stores this token (usually in localStorage or sessionStorage) and sends it along with each subsequent request that requires authentication.

Example Request

Json{

"email": "John doe@example.com"

"password": "securepassword123"

}

"Example Response

{

"token"

}

3. Token Storage & Usage:

The soken is typically stored in the browser's localStorage or session Storage. Every subsequent request to a protected route will include the token in the HTTP Authorization header.

Authorization: Bearer CWT>

The server then venifies this token to authenticate the user before granting access to the requested resource.

JWT Token Structure:

The JWT contains three parts:

1. "Header: Information about the token's algorithm (eg, H5256).

2. "Payload: The user's data (e.g., user ID, role, expiration time).

3. "Signature": A secure hash to verify that the token has not been tampered with.

2. Authorization in MERN Application

"What is Authorization?

Authorization refers to the process of determining what a user is allowed to do after their Identity has been authenticated. It ensures that only authorized users can access specific resources or perform certain actions (eg., only a freelancer can place a bid, only an admin can block users).

"How Authorization Works:"

Once the user is authenticated with a JWT token, the server checks the user's "role" and "permissions" to determine whether they can access a resource or perform an action.

1. "Role-Based Authorization:*

A common method of authorization in freelancing applications is "role-based access control (RBAC).

USER INTERFACE :

User Interface (UI) Design for Freelancing Application - MERN

The User Interface (UI) of the freelancing application plays a crucial role in providing a seamless and intuitive experience for both freelancers and clients. Below is a detailed description of the UI design and structure for the freelancing application built with the MERN stack (MongoDB, Express, React, Node.js). This explanation will cover the layout, components, and user flows for both freelancers and clients.

1. User Types:

There are two main types of users in the freelancing application:

- Freelancer: A user offering services and applying for jobs.
- Client: A user posting job listings and hiring freelancers.

The UI design should adapt to both user types with different experiences and functionalities available for each.

2. Main Pages & Components

a. Homepage (Public Access)

The homepage is the first point of interaction for both freelancers and clients. It provides basic information about the platform and access to login or registration.

- Header:
  - Navigation links: Home, About, Login, Register.

- Logo and branding of the freelancing platform.
  - Optional: Language or theme selection.
- Main Content:
  - Hero Section: Introduction to the platform with a brief description of services for freelancers and clients.
  - Call-to-Action (CTA): Buttons such as “Find a Freelancer” (for clients) and “Find Jobs” (for freelancers).
  - Features Section: Highlights the key benefits of the platform.
  - Footer: Links to Privacy Policy, Terms of Service, and Contact Us.

b. Registration & Login Page

For both freelancers and clients, users must first register to use the platform. The registration page contains the following:

- Form Elements:
  - Name, Email, Password, and Role (Freelancer or Client).
  - Password strength indicator and validation.

- “Submit” button to register the user.

Login Page:

- Form Elements:
  - Email and Password fields.
  - Login button to authenticate users.
  - Option to Reset Password or Go to Register for new users.

- Responsive design: The registration and login forms should be mobile-friendly, with fields stacked on smaller screens.

c. Dashboard (Freelancer and Client)

Once logged in, both freelancers and clients are directed to their respective dashboards, where they can access their key functionalities.

Freelancer Dashboard:

- Header:
  - Navigation bar with Home, Profile, Job Feed, Messages, and Logout.
- Main Content:

  - Job Feed: Displays a list of available job postings from clients. Freelancers can search and filter jobs based on categories, budget, and job title.

  - Profile Section: Shows freelancer's profile details (name, experience, skills, rating, etc.). Includes an option to edit the profile.

  - Application Status: Displays the status of applied jobs, whether Accepted, Rejected, or Pending.

- Sidebar:

  - Quick access to job applications, messages, and notifications.

Client Dashboard:

- Header:

  - Navigation bar with Home, Post a Job, My Jobs, Messages, and Logout.

- Main Content:

  - Job Posting: Clients can view the jobs they've posted. Each job posting shows the number of applicants, status, and job details.

  - Post a Job: Option to post new jobs. Clients can fill in fields such as job title, description, budget, and deadline.

- Freelancer Applications: Clients can view the freelancers who have applied for their jobs, view profiles, and select freelancers.

- Sidebar:

  - Quick access to manage posted jobs, job applications, and messaging.

Job Posting (for Clients):

- A form to create a job listing with input fields like title, description, budget, skills required, deadline, and job category.

- Preview Option: Before posting, clients can preview the job description to ensure it's correct.

- After posting, the job will appear in the job feed for freelancers to apply.

d. Profile Page

Freelancer Profile:

- Personal Details: Name, profile picture, bio, contact information.

- Skills & Experience: A list of services offered, skills, education, and work experience.

- Portfolio: A section for freelancers to showcase their previous work or projects.

- Ratings & Reviews: A section where clients can rate freelancers after completing a job.

Client Profile:

- Personal/Business Details: Name, business name, contact info, and profile picture.

- Job Postings: A history of the client's posted jobs.

- Ratings: A section where freelancers can leave feedback for clients after working with them.

f. Notifications

- System Notifications: Freelancers and clients can receive alerts for new job postings, job application status updates, messages, and job acceptance.

- Notification Center: A bell icon or a dedicated section where users can check their notifications.

3. Mobile Responsiveness

The UI must be responsive and adapt to different screen sizes. Key considerations include:

- Mobile Navigation: Use a hamburger menu for easy access to different pages (Home, Profile, Jobs, Messages).

- Form Layouts: Stack input fields vertically on smaller screens.

- Job Feed: Display jobs in a list format that can be scrolled vertically.

4. UI Design Elements

- Colors: Use a professional and clean color scheme, with contrast for readability (e.g., blue for buttons, white background with dark text).

- Typography: Use clear and legible fonts (e.g., Roboto, Open Sans).

- Buttons and Icons: Use call-to-action buttons (e.g., "Apply", "Post Job") that stand out. Icons should be used for common actions (e.g., Message icon, Settings icon).

- User Feedback: Show success/error messages in clearly visible areas (e.g., form validation, job application success).

5. User Flows

- Freelancer Flow:

1. Visit Home → Sign Up → Browse Job Feed → Apply for a job → Wait for client response → Manage applications → View messages.

- Client Flow:

1. Visit Home → Sign Up → Post a Job → Manage Job Posts → Review freelancer applications → Select a freelancer → View messages.

6. Tools Used in UI Development

- React for building the UI components.
- React Router for navigation between pages (e.g., Home, Login, Dashboard, Profile).
- Axios for making API requests to the backend.

This design description will help in explaining the different components and pages of the freelancing application in your project report. Let me know if you need any further clarifications or additional details!

TESTING:

Testing in Freelancing Application (MERN)

Testing is a critical part of any web application development process, ensuring that the system functions as expected and that users have a seamless experience. For the Freelancing Application built with MERN (MongoDB, Express, React, Node.js), testing can be divided into different categories based on the functionality and type of testing required. Below are the types of testing along with diagrammatic representations to provide a clear understanding of the testing process.

1. Types of Testing

a. Unit Testing

Unit testing focuses on testing individual units or components of the application in isolation. For a MERN application, it would typically involve testing the backend API routes, React components, and utility functions.

[React Component] --> [Test React Component] --> [Backend API Routes] --> [Utility Functions]

b. Integration Testing

Integration testing ensures that multiple components of the application work together as expected. In a MERN application, this involves testing the interaction between the backend and frontend, such as making sure the React frontend correctly interacts with the Express backend and retrieves data from MongoDB.

[React Frontend] --> [API Calls (Axios)] --> [Express Backend] --> [MongoDB Database]

c. Functional Testing

Functional testing verifies that the application’s features and functionalities perform as expected. For example, testing whether a user can log in, post a job, or apply for a job correctly.

[User Interactions] --> [React Components] --> [Express API Routes] --> [MongoDB Operations]

d. End-to-End (E2E) Testing

E2E testing validates the entire flow of the application, simulating a real-world user interaction from start to finish. For a freelancing application, this would test entire workflows such as user registration, job posting, and freelancing job applications.

[Test User Registration] --> [Backend API Register] --> [Test Login Process] --> [Test Job Posting Flow] --> [Test Job Application]

e. Performance Testing

Performance testing ensures that the application performs efficiently under load, including checking the server's response time, how the frontend reacts under heavy traffic, and whether the system can handle multiple concurrent users.

[Load Testing] --> [Backend Load (API Requests)] --> [Database Load Testing]

2. Testing Tools

Each type of testing can be done using different tools:

- Unit Testing:
  - Jest or Mocha for backend.
  - React Testing Library for frontend components.

- Integration Testing:

- Supertest for API integration tests (testing API routes with Express).
  - Jest for testing frontend-backend interactions.

- Functional Testing:
  - Selenium for simulating user interactions.
  - Cypress for testing UI behaviors like form submission, validation, and navigation.

- End-to-End (E2E) Testing:
  - Cypress or Puppeteer for running complete workflows to validate entire user journeys.

- Performance Testing:
  - JMeter for simulating user load on the server.
  - Lighthouse for checking frontend performance (load times, rendering speed).

3. Testing Workflow Overview

Below is an overall testing workflow for the freelancing application:

```
[Start Development]
       |
       v
[Unit Tests]
       |
       v
[Integration Tests]
       |
       v
[Functional Tests]
       |
       v
[End-to-End Tests]
       |
       v
[Performance Tests]
       |
       v
[Final Deployment]
```

This approach provides a comprehensive testing strategy to ensure the platform works seamlessly for both freelancers and clients, improving the overall quality and user satisfaction.

SCREENSHOTS OR DEMO :

For the freelancing application MERN project that you're working on, here's a detailed description of the screenshots and demo that would typically be included to showcase the project:

1.Landing Page Layout:

The Landing Page for your freelancing application built with the MERN stack will play a critical role in attracting users (both freelancers and employers) and guiding them to take action .It containes

Logo: Positioned on the top left, representing the brand of the freelancing platform.

Home

About Us

Features

How It Works

Contact Us

2. Login/Sign-up Page:

Screenshot: A clean interface with fields for email/username and password.

Features to Highlight:

User authentication (sign-up, sign-in, password recovery).

User roles (Freelancer, Employer).

3. New Project Page:

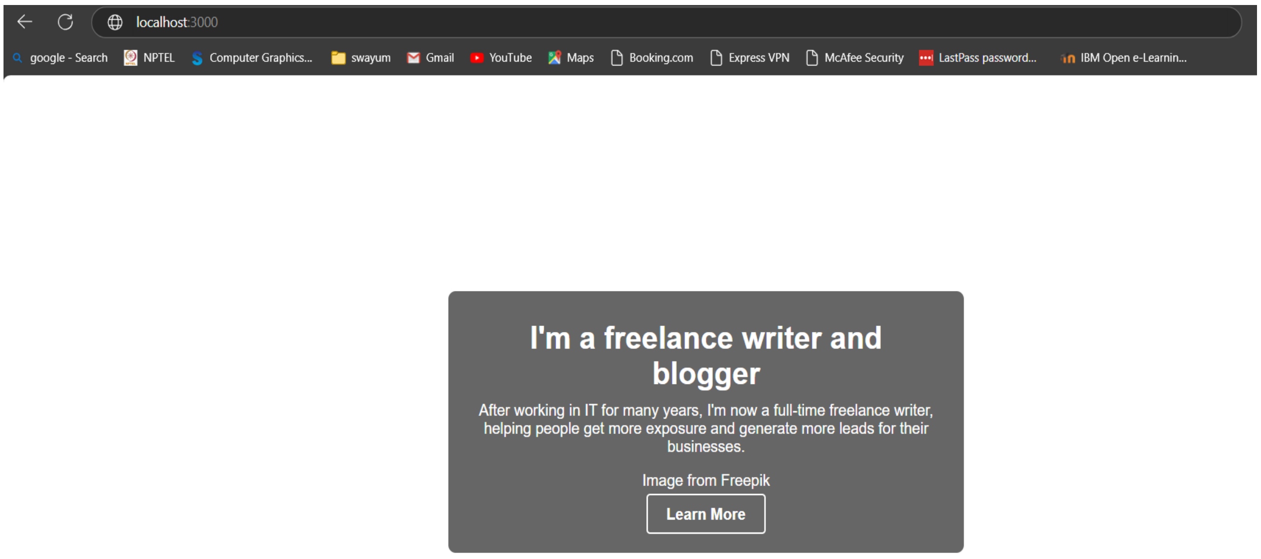
Title of the Form: “Post a New Job” or “Create a New Project”

Form Fields:

Job Title: A short and clear title for the job (e.g., "Web Developer for E-commerce Site").

Job Description: A large text area where employers can describe the job in detail, including project scope, deadlines, and requirements.

Budget: Input field for entering the budget or pay rate (can be hourly or fixed price).

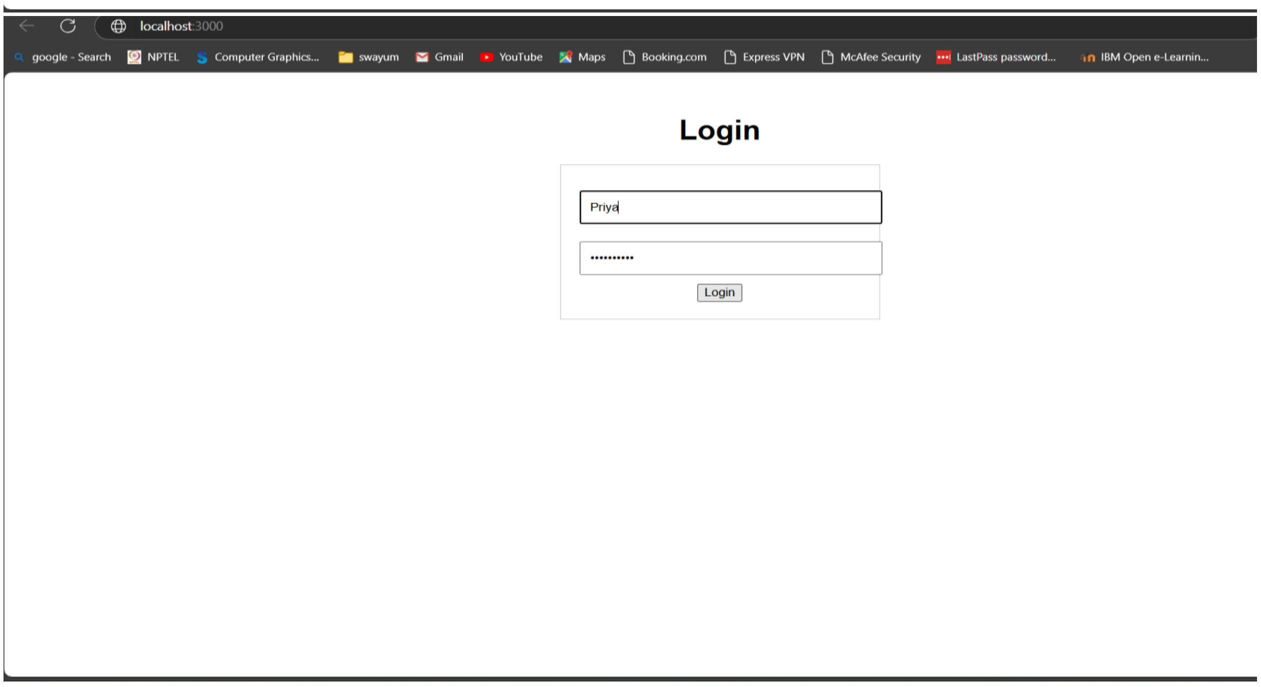

localhost:3000

google - Search, NPTEL, Computer Graphics..., swayum, Gmail, YouTube, Maps, Booking.com, Express VPN, McAfee Security, LastPass password..., IBM Open e-Learnin...

I'm a freelance writer and blogger

After working in IT for many years, I'm now a full-time freelance writer, helping people get more exposure and generate more leads for their businesses.

Image from Freepik

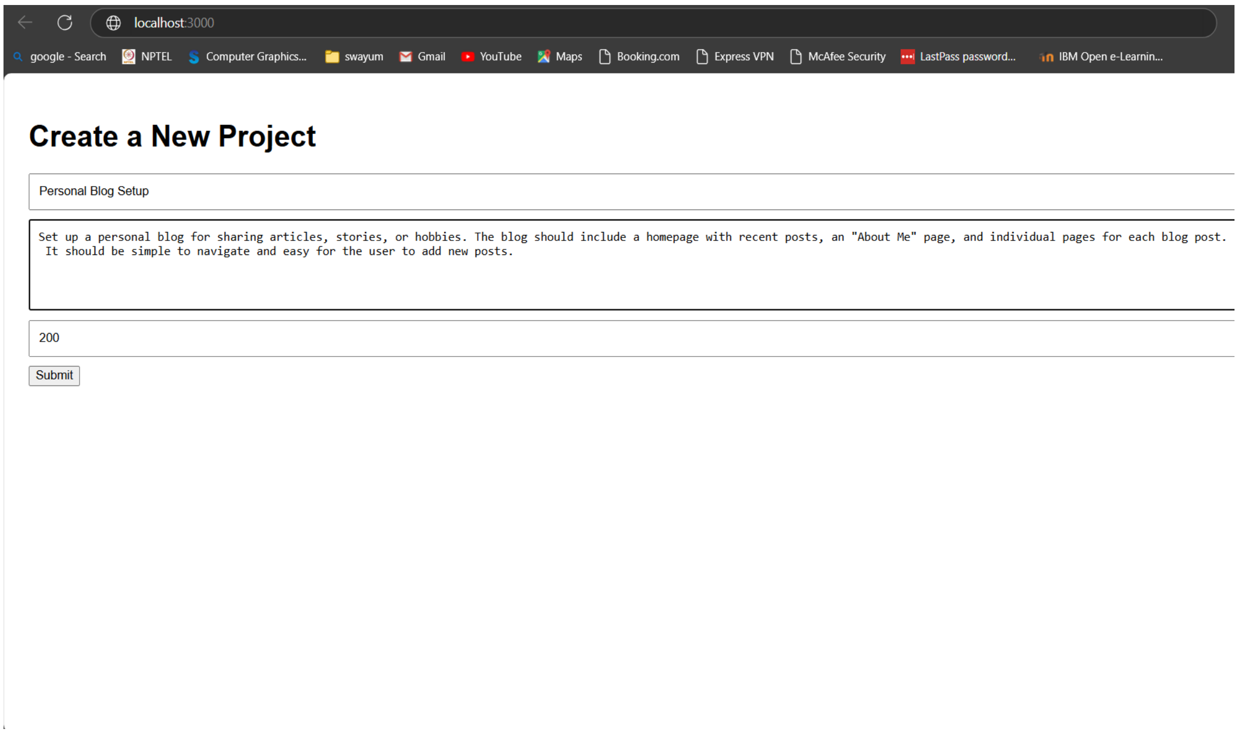

localhost:3000

google - Search, NPTEL, Computer Graphics..., swayum, Gmail, YouTube, Maps, Booking.com, Express VPN, McAfee Security, LastPass password..., IBM Open e-Learnin...

Login

Priya

••••••••••


localhost:3000

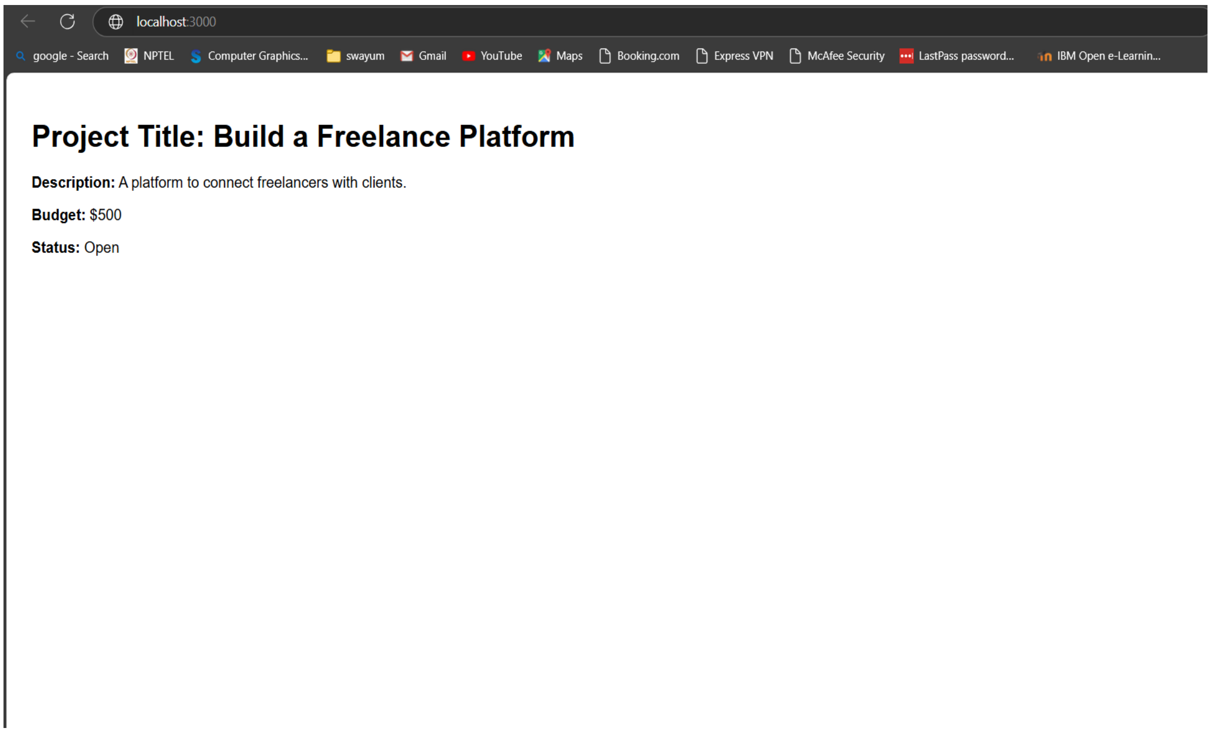
google - Search | NPTEL | Computer Graphics... | swayum | Gmail | YouTube | Maps | Booking.com | Express VPN | McAfee Security | LastPass password... | IBM Open e-Learnin...

Create a New Project

Personal Blog Setup

Set up a personal blog for sharing articles, stories, or hobbies. The blog should include a homepage with recent posts, an "About Me" page, and individual pages for each blog post. It should be simple to navigate and easy for the user to add new posts.

200


localhost:3000

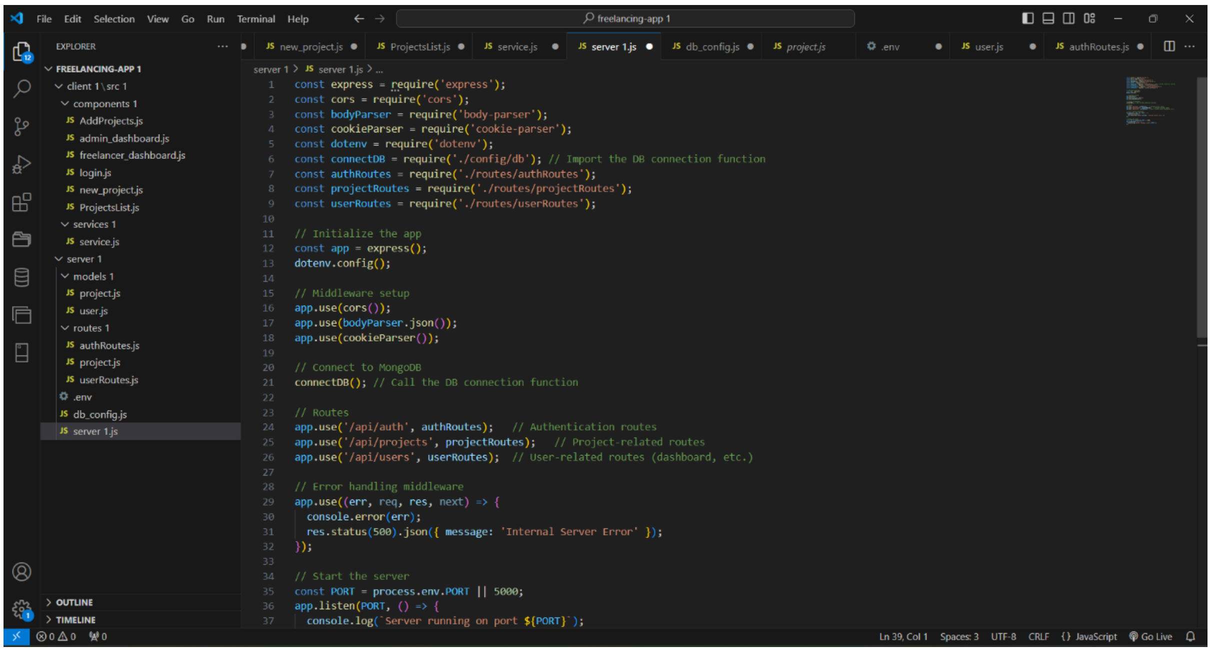
google - Search | NPTEL | Computer Graphics... | swayum | Gmail | YouTube | Maps | Booking.com | Express VPN | McAfee Security | LastPass password... | IBM Open e-Learnin...

Project Title: Build a Freelance Platform

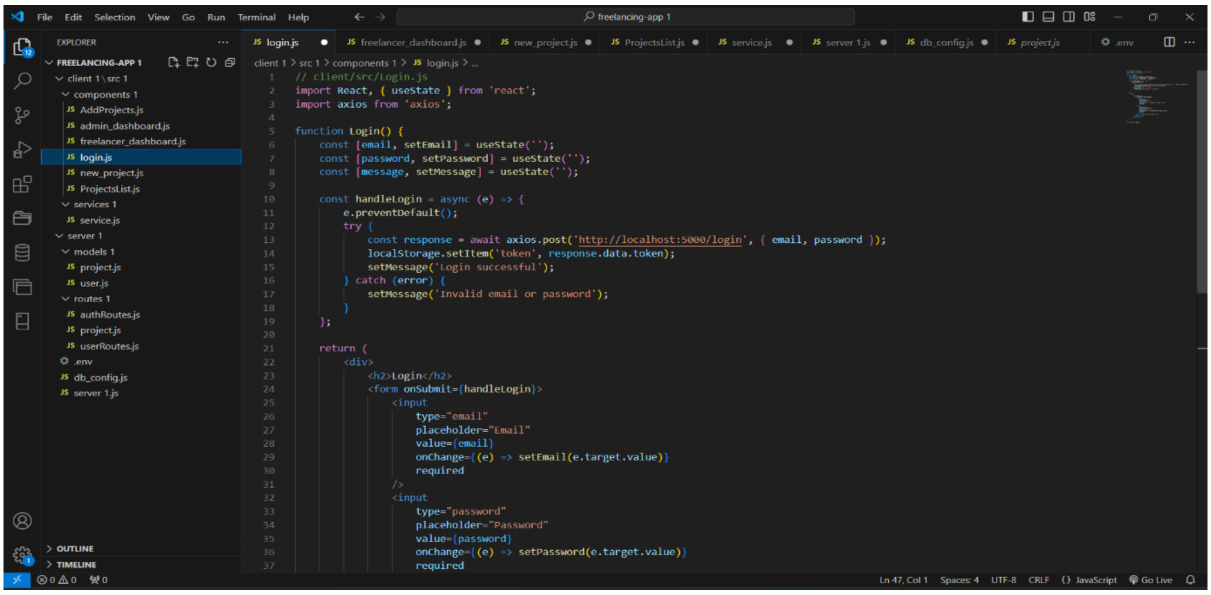
Description: A platform to connect freelancers with clients.

Budget: $500

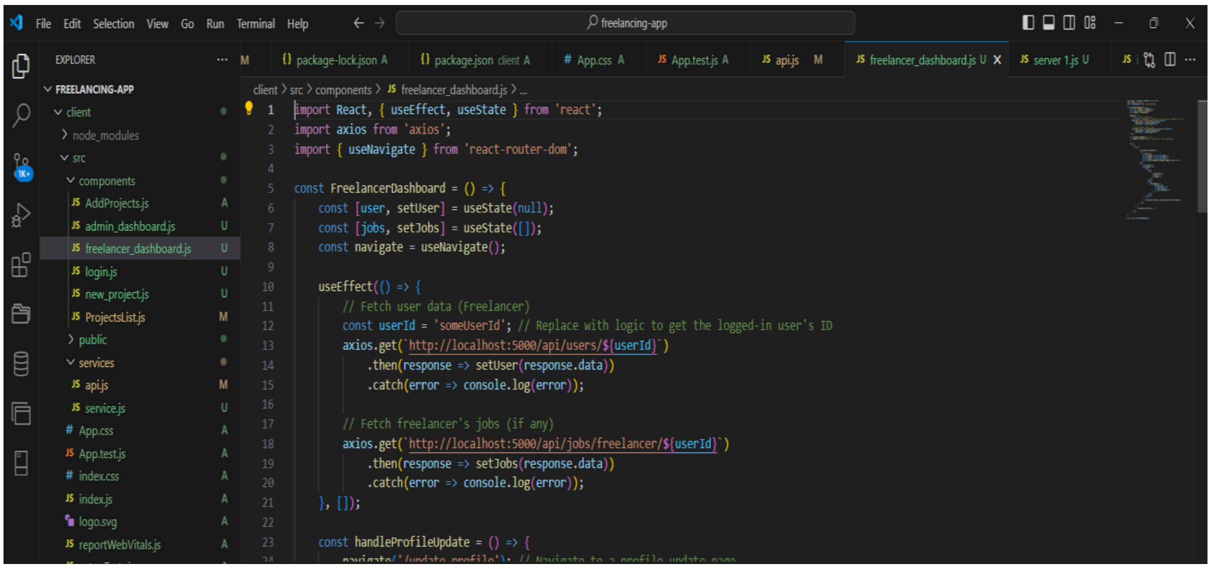
Status: Open



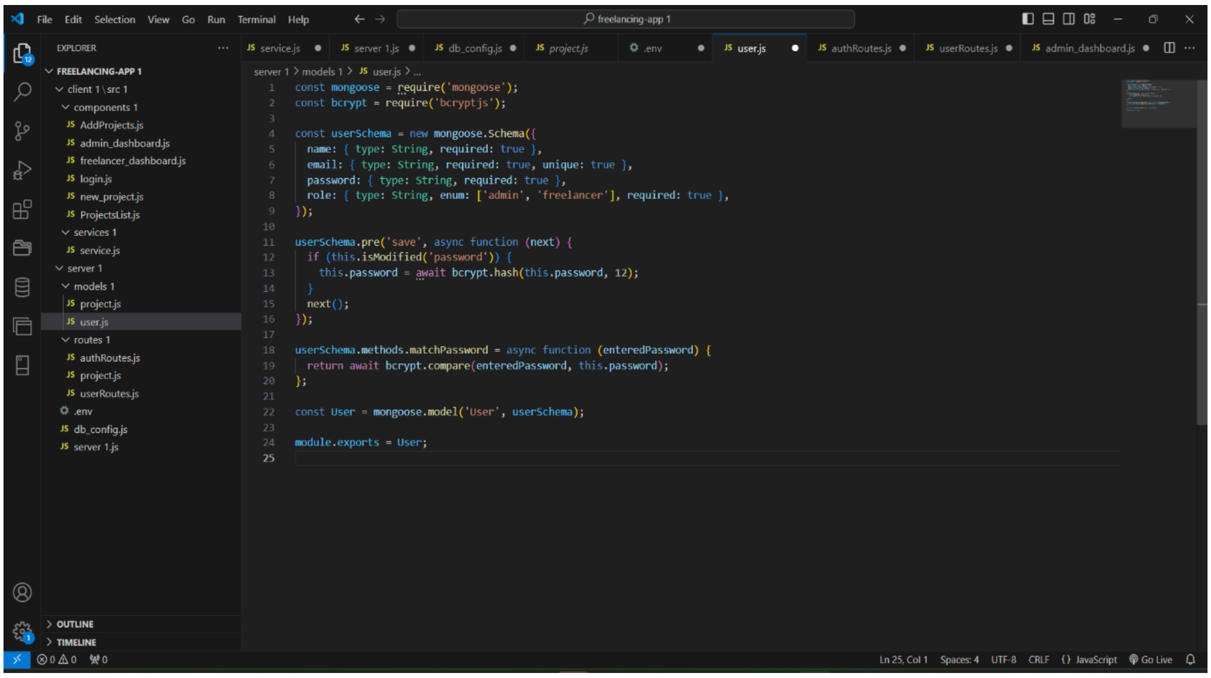
```javascript
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const dotenv = require('dotenv');
const connectDB = require('./config/db'); // Import the DB connection function
const authRoutes = require('./routes/authRoutes');
const projectRoutes = require('./routes/projectRoutes');
const userRoutes = require('./routes/userRoutes');

// Initialize the app
const app = express();
dotenv.config();

// Middleware setup
app.use(cors());
app.use(bodyParser.json());
app.use(cookieParser());

// Connect to MongoDB
connectDB(); // Call the DB connection function

// Routes
app.use('/api/auth', authRoutes);   // Authentication routes
app.use('/api/projects', projectRoutes);   // Project-related routes
app.use('/api/users', userRoutes);  // User-related routes (dashboard, etc.)

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ message: 'Internal Server Error' });
});

// Start the server
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
```



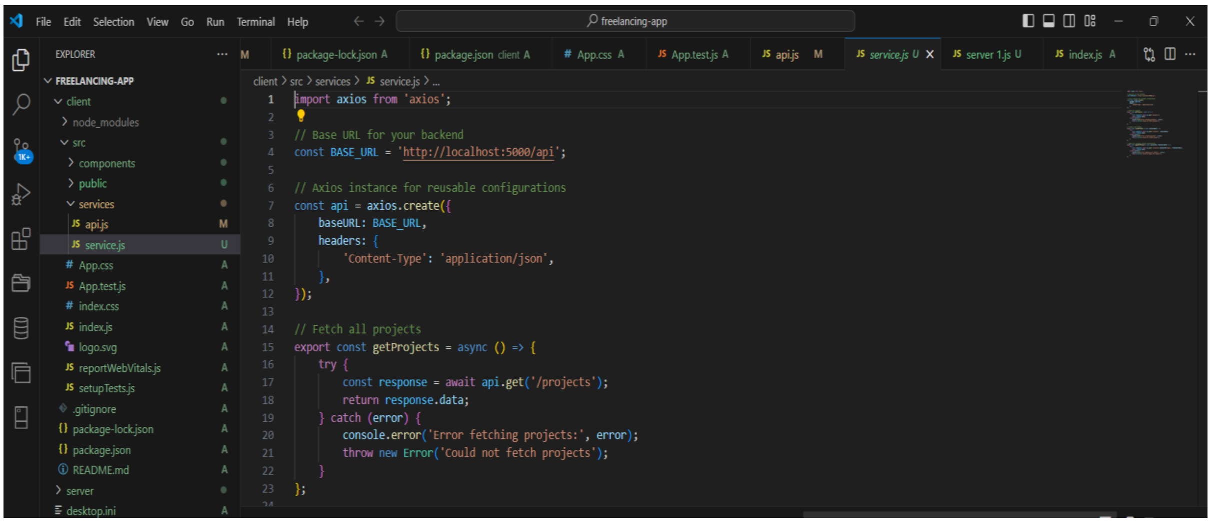
```javascript
// client/src/Login.js
import React, { useState } from 'react';
import axios from 'axios';

function Login() {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const [message, setMessage] = useState('');

    const handleLogin = async (e) => {
        e.preventDefault();
        try {
            const response = await axios.post('http://localhost:5000/login', { email, password });
            localStorage.setItem('token', response.data.token);
            setMessage('Login successful');
        } catch (error) {
            setMessage('Invalid email or password');
        }
    };

    return (
        <div>
            <h2>Login</h2>
            <form onSubmit={handleLogin}>
                <input
                    type="email"
                    placeholder="Email"
                    value={email}
                    onChange={(e) => setEmail(e.target.value)}
                    required
                />
                <input
                    type="password"
                    placeholder="Password"
                    value={password}
                    onChange={(e) => setPassword(e.target.value)}
                    required
```



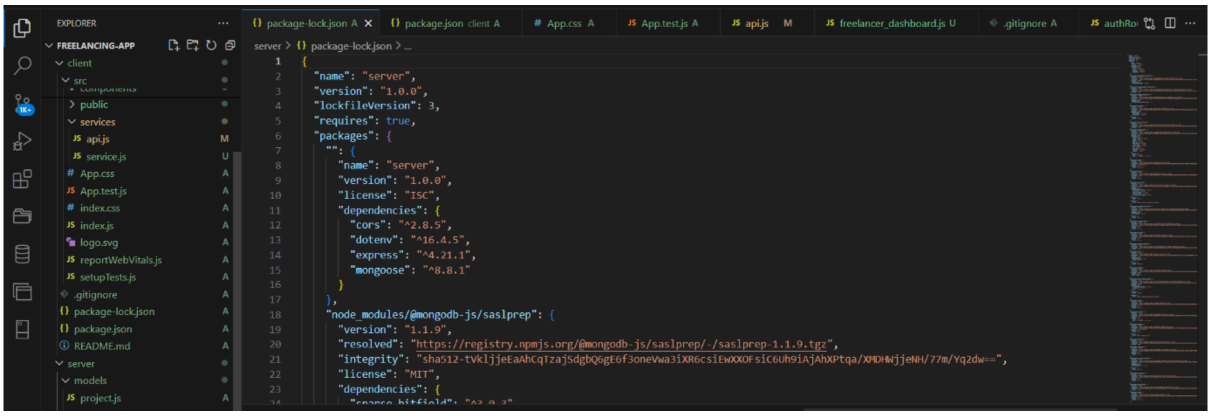
```javascript
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import { useNavigate } from 'react-router-dom';

const FreelancerDashboard = () => {
    const [user, setUser] = useState(null);
    const [jobs, setJobs] = useState([]);
    const navigate = useNavigate();

    useEffect(() => {
        // Fetch user data (Freelancer)
        const userId = 'someUserId'; // Replace with logic to get the logged-in user's ID
        axios.get(`http://localhost:5000/api/users/${userId}`)
            .then(response => setUser(response.data))
            .catch(error => console.log(error));

        // Fetch freelancer's jobs (if any)
        axios.get(`http://localhost:5000/api/jobs/freelancer/${userId}`)
            .then(response => setJobs(response.data))
            .catch(error => console.log(error));
    }, []);

    const handleProfileUpdate = () => {
        navigate('/update_profile'); // Navigate to a profile update page
```



```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ['admin', 'freelancer'], required: true },
});

userSchema.pre('save', async function (next) {
  if (this.isModified('password')) {
    this.password = await bcrypt.hash(this.password, 12);
  }
  next();
});

userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};

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

module.exports = User;
```



```
import axios from 'axios';

// Base URL for your backend
const BASE_URL = 'http://localhost:5000/api';

// Axios instance for reusable configurations
const api = axios.create({
    baseURL: BASE_URL,
    headers: {
        'Content-Type': 'application/json',
    },
});

// Fetch all projects
export const getProjects = async () => {
    try {
        const response = await api.get('/projects');
        return response.data;
    } catch (error) {
        console.error('Error fetching projects:', error);
        throw new Error('Could not fetch projects');
    }
};
```



```
{
  "name": "server",
  "version": "1.0.0",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "name": "server",
      "version": "1.0.0",
      "license": "ISC",
      "dependencies": {
        "cors": "^2.8.5",
        "dotenv": "^16.4.5",
        "express": "^4.21.1",
        "mongoose": "^8.8.1"
      }
    },
    "node_modules/@mongodb-js/saslprep": {
      "version": "1.1.9",
      "resolved": "https://registry.npmjs.org/@mongodb-js/saslprep/-/saslprep-1.1.9.tgz",
      "integrity": "sha512-tVkljjeEaAhCqTzajSdgbQ6gE6f3oneVwa3iXR6csiEwXXOFsiC6Uh9iAjAhXPtqa/XMDHWjjeNH/77m/Yq2dw==",
      "license": "MIT",
      "dependencies": {
```

KNOWN ISSUES :

When developing a Freelancing Application using the MERN stack, there are a few common challenges and known issues that may arise during both development and deployment. Below are some of the key known issues for such an application:

1. Authentication and Authorization Problems:

- **Issue**: Users unable to log in or facing issues with registration.
- **Cause**:
  - Incorrect JWT token handling or token expiration.
  - Issues with password hashing or bcrypt.
  - Incorrect configuration of session management (e.g., cookie settings for JWT).
- **Solution**:
  - Ensure that token expiration is properly set and handled.
  - Use libraries like **bcryptjs** for hashing passwords securely.
  - Test the flow thoroughly for both authentication and authorization.

2. Database Connection Issues (MongoDB):

- **Issue**: Application fails to connect to the MongoDB database or experiences slow queries.
- **Cause**:
    - Incorrect database connection URL (e.g., wrong credentials or database name).
    - Overuse of resources or improper indexing in the MongoDB collections.
- **Solution**:
    - Double-check the **MongoDB URI** and ensure it is correct.
    - Optimize your queries and add indexes for faster lookups on frequently queried fields (e.g., job title, freelancer skills).
    - Ensure that MongoDB is properly hosted or configured (e.g., using MongoDB Atlas for cloud hosting).

3. Frontend UI/UX Issues (React):

- **Issue**: Pages are slow to load, or the UI elements don't display correctly.
- **Cause**:

    - **Large images** or **unoptimized assets** (e.g., CSS, JavaScript).
    - **Improper state management** causing slow re-renders (e.g., unnecessary re-renders in React components).
- **Solution**:
    - Optimize images and other assets (use **image compression tools**).
    - Use **React.memo** or **useMemo** to prevent unnecessary re-renders.
    - Implement lazy loading for large components and external resources.

4. Job Listing and Filtering Problems:

- **Issue**: Job search or filtering options aren't functioning correctly.
- **Cause**:
    - Issues with query parameters in the **backend** (e.g., improper handling of filters for job categories, budgets, and locations).
    - Frontend not sending the correct request to the backend for filtering.
- **Solution**:

    - Ensure the **backend** correctly handles query parameters and returns accurate results.
    - On the **frontend**, ensure that all filter options are properly mapped to query parameters.

5. Real-Time Communication (Chat) Not Working:

- **Issue**: The messaging/chat feature doesn't update in real-time or messages are delayed.
- **Cause**:
    - **WebSocket** or **Socket.IO** connection issues.
    - Poor event handling or message delivery failures.
- **Solution**:
    - Ensure **Socket.IO** is properly configured on both the **client** and **server**.
    - Implement event listeners for proper message flow and handling.
    - Check for any WebSocket connection limits or conflicts in the server configuration.

6. Payment Gateway Integration Issues:

- **Issue**: Freelancers or employers can't complete transactions, or payments aren't processed.
- **Cause**:

    - Incorrect integration of payment gateways like **PayPal**, **Stripe**, etc.
    - Failure to handle **payment confirmation** or **transaction failures**.
- **Solution**:
    - Double-check the integration of payment APIs and ensure you handle errors gracefully (e.g., failed payments).
    - Use **webhooks** to listen for payment status updates and process them on the backend.
    - Test the payment flow using sandbox accounts provided by the payment gateways (Stripe, PayPal).

FUTURE ENHANCEMENTS :

future enhancements for your **Freelancing Application MERN** that could improve its functionality, scalability, and user experience:

1. Advanced Search and Filtering Options

- **Feature**: Enhance the job search functionality by adding more advanced filters.
- **Details**: Allow freelancers and employers to filter jobs based on more criteria like:

    - Experience level (Entry-level, Intermediate, Expert)
    - Job duration (Short-term, Long-term, Contract)
    - Skills/technologies required (e.g., React, Node.js, Python)
    - Location-based filtering (Remote jobs, or region-specific)
- **Benefit**: Improves user experience by making it easier to find relevant opportunities or freelancers.

2. Rating and Review System

- **Feature**: Implement a robust rating and review system for freelancers and employers.
- **Details**: After each completed job, both freelancers and employers can rate each other based on the quality of the work, communication, and timeliness. This system could include:
    - Star ratings (1-5 stars)
    - Written feedback
- **Benefit**: Builds trust within the community by allowing users to make more informed decisions based on previous feedback.

3. Job Matching Algorithm

- **Feature**: Develop an intelligent matching algorithm that automatically suggests jobs to freelancers and candidates to employers.
- **Details**: The algorithm can use machine learning (ML) or predefined logic to match the job requirements with freelancer skills, experience, and location preferences.
- **Benefit**: Saves time for both freelancers and employers by surfacing the best matches automatically, increasing the likelihood of successful collaborations.

4. In-App Video Conferencing and Voice Calls

- **Feature**: Add in-app video and voice calling features for better communication between freelancers and employers.
- **Details**: Integrate tools like WebRTC or third-party APIs (e.g., Zoom, Twilio) to allow users to have face-to-face meetings or voice calls within the app.
- **Benefit**: Eliminates the need for external communication tools, making the process smoother and more integrated.

5. Escrow Payment System

- **Feature**: Implement an escrow payment system to hold funds in trust until both parties are satisfied.

- **Details**: Employers can deposit the payment into escrow before the job begins, and the funds are only released when the freelancer completes the job to the employer's satisfaction.
- **Benefit**: Ensures security for both parties—freelancers are assured of payment, and employers are protected from incomplete or subpar work.

6. Mobile App Development

- **Feature**: Develop native mobile apps for iOS and Android to allow users to access the freelancing platform on the go.
- **Details**: Create mobile versions of the freelancing app using React Native or other mobile development frameworks.
- **Benefit**: Increases user engagement, as many freelancers and employers prefer using mobile devices to manage jobs and communications.

7. Multi-Language and Multi-Currency Support

- **Feature**: Implement multi-language and multi-currency support for global users.
- **Details**: Allow users to choose their preferred language and currency, making the platform more accessible to a global audience.

- **Benefit**: Expands the platform’s reach and usability to international markets.