FreelanceHub – A MERN Stack Freelancing Platform

Team:

Mekapothula Venkata Pavan – Full Stack Developer (Frontend + Backend + Database Integration)

Register Number:SBAP0069787

Email :venkatpavan165@gmail.com

Mobile Number :9491557474

1. Introduction

FreelanceHub is a full-stack web application developed using the **MERN stack (MongoDB, Express.js, React.js, Node.js)**. The platform connects clients and freelancers, enabling them to collaborate on projects efficiently and securely.

The system allows clients to post projects, freelancers to bid on them, and both parties to manage communication and payments within the platform. The application is designed to provide a seamless user experience, secure authentication, real-time updates, and scalable architecture.

This project demonstrates the implementation of modern web development practices including:

- RESTful API development
- JWT-based authentication
- Role-based access control
- Responsive frontend design
- CRUD operations with MongoDB

2. Project Overview

Purpose

The primary purpose of **FreelanceHub – MERN Freelancing Platform** is to create a secure and efficient digital marketplace where clients and freelancers can connect, collaborate, and complete projects seamlessly.

The goals of the project are:

- To provide a centralized platform for posting and managing freelance projects.
- To enable freelancers to explore opportunities and submit bids.
- To implement secure authentication and role-based authorization.
- To ensure smooth communication and project tracking between users.
- To demonstrate real-world full stack development using the MERN stack.

This project aims to simulate a real-world freelancing ecosystem similar to platforms like Upwork or Fiverr while implementing modern web technologies and scalable architecture.

Features

The FreelanceHub platform offers a comprehensive set of features designed to facilitate seamless interaction between clients and freelancers. The system ensures secure communication, efficient project management, and an intuitive user experience.

The application implements secure user authentication using JSON Web Tokens (JWT). Users can register and log in securely, and passwords are encrypted using hashing techniques before being stored in the database. Protected routes ensure that only authenticated users can access authorized resources, maintaining system security and data integrity.

The platform supports role-based access control, allowing different types of users such as clients and freelancers to access specific functionalities according to their responsibilities. This structured access mechanism prevents unauthorized actions and enhances overall system reliability.

Clients can create and manage projects by providing details such as title, description, required skills, budget, and deadline. The system allows modification or deletion of projects and enables status updates such as Open, In Progress, and Completed. This ensures transparency and efficient project tracking.

Freelancers can browse available projects and submit proposals that include bid amount, estimated completion time, and a brief message describing their approach. Clients can review these proposals and select suitable freelancers based on skills, experience, and budget considerations.

The application provides personalized dashboards that display relevant information to users. Freelancers can monitor their submitted bids and track project progress, while clients can manage posted projects and review incoming proposals. These dashboards dynamically update through backend API integration.

Users can manage and update their profile information, including personal details, skills, and experience. Profile management enhances credibility and helps improve trust between clients and freelancers.

The backend follows RESTful API architecture using Node.js and Express.js. All operations such as authentication, project creation, proposal submission, and profile updates are handled through structured API endpoints, ensuring scalability and maintainability.

MongoDB is used as the database system to store user information, project data, and bid details. The database schema is designed to allow efficient querying, scalability, and secure data storage.

The frontend is developed using React.js with a component-based architecture. The user interface is responsive and adaptable across various devices, ensuring consistent user experience on desktops, tablets, and mobile devices.

3. Architecture

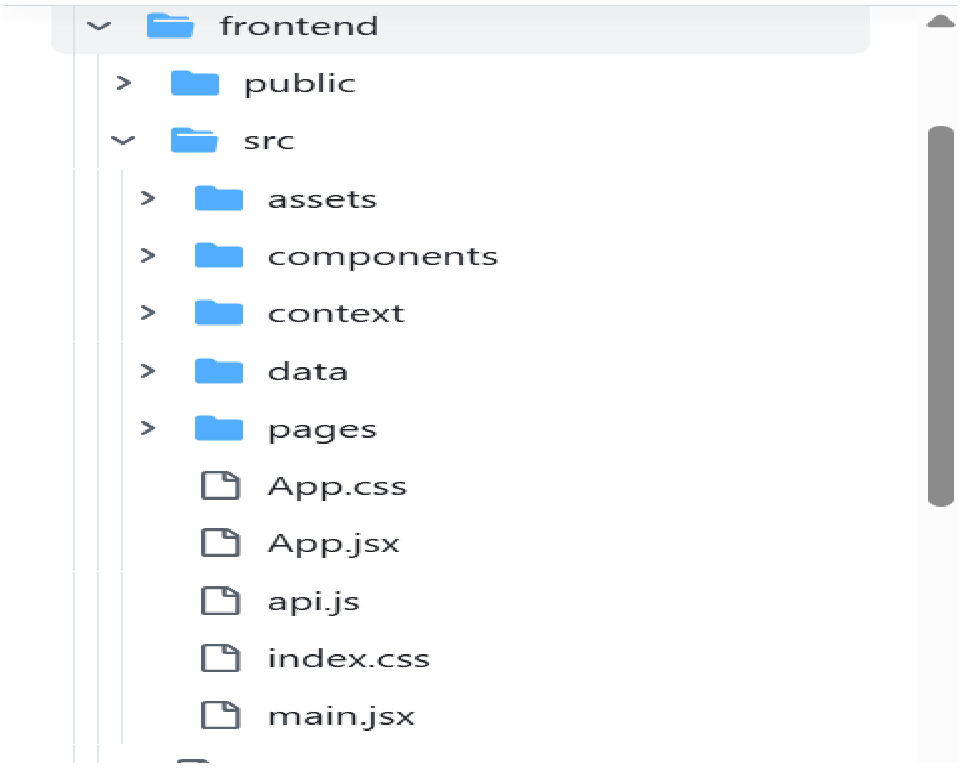
The FreelanceHub platform follows a three-tier architecture consisting of the Frontend (Client Side), Backend (Server Side), and Database Layer. The system is designed to ensure scalability, maintainability, and clear separation of concerns.

Frontend Architecture (React.js)

The frontend of the application is developed using React.js and is located inside the frontend directory. The structure is modular and component-based, allowing efficient development and maintenance.

Frontend Folder Structure

- **public/**
  Contains static files such as the HTML template and favicon.
- **src/**
  Main source directory containing all React application logic.
  - **assets/**
    Stores images, icons, and other static resources used in the UI.



```
frontend
  public
  src
    assets
    components
    context
    data
    pages
    App.css
    App.jsx
    api.js
    index.css
    main.jsx
```

-
  - **components/**
    Contains reusable UI components such as Navbar, ProtectedRoute, Cards, Buttons, etc. These components promote code reusability and maintainability.
  - **context/**
    Manages global state using React Context API (such as authentication state and user session data).
  - **data/**
    Stores static or mock data used within the application.
  - **pages/**
    Contains page-level components such as:
    - Login
    - Register
    - Dashboard

        - Project Pages
        - Profile Pages
          Each page represents a route in the application.

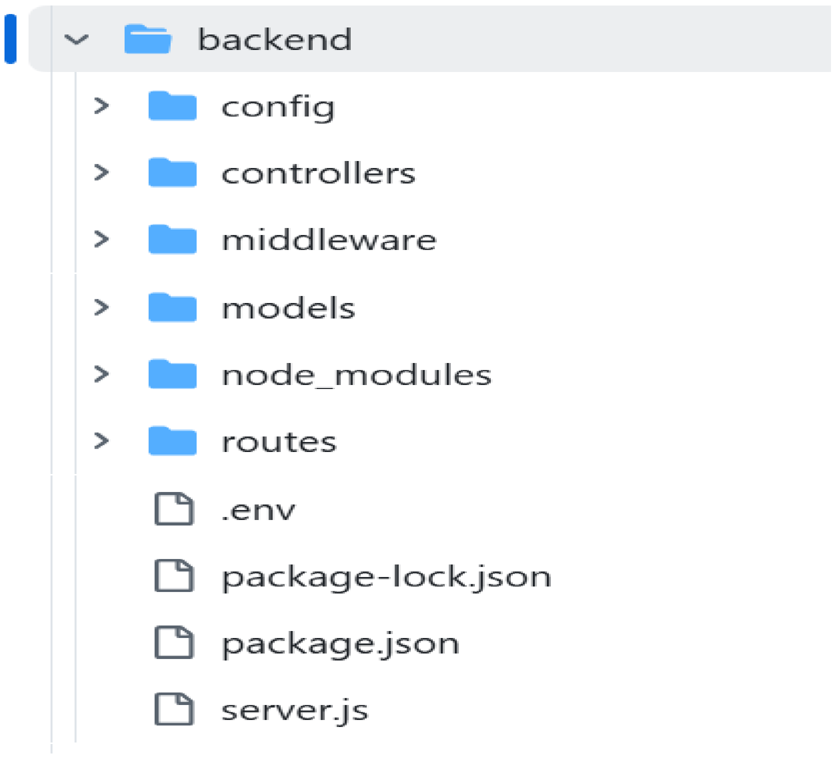
    - **App.jsx**
      Defines the main routing structure using React Router and controls navigation between pages.
    - **main.jsx**
      Entry point of the React application that renders the root component.
    - **api.js**
      Handles API requests to the backend server using Axios or Fetch API.
    - **App.css / index.css**
      Contains global and component-level styling.

Frontend Working Flow

1. User interacts with UI components.
2. API calls are made through api.js.
3. Data is fetched from the backend.
4. State is updated using Context API or React hooks.
5. UI dynamically re-renders based on updated state.

This architecture ensures clean code organization and smooth communication with the backend.

Backend Architecture (Node.js & Express.js)



```
backend
    config
    controllers
    middleware
    models
    node_modules
    routes
    .env
    package-lock.json
    package.json
    server.js
```

The backend is developed using Node.js and Express.js and is located inside the backend directory. It follows a modular MVC (Model-View-Controller) architecture.

Backend Folder Structure

- **config/**
  Contains database configuration and environment setup files.
- **controllers/**
  Contains business logic functions that handle incoming requests and return responses.
- **middleware/**
  Includes authentication middleware (JWT verification), error handling, and authorization checks.
- **models/**
  Defines MongoDB schemas using Mongoose.
  Example models:
  - User Model
  - Project Model
  - Bid Model
- **routes/**
  Defines API endpoints and connects them to corresponding controller functions.
- **server.js**
  Entry point of the backend application.
  Initializes Express app, connects to MongoDB, sets up middleware, and defines route usage.
- **.env**
  Stores environment variables such as:
  - MongoDB URI
  - JWT Secret Key
  - Port number
- **package.json**
  Manages project dependencies and scripts.

Backend Working Flow

1. Client sends HTTP request.
2. Route handles request.
3. Middleware verifies authentication (if required).
4. Controller executes business logic.
5. Model interacts with MongoDB.
6. Response is sent back as JSON.

This layered structure improves maintainability, scalability, and security.

Database Connection Configuration

The application connects to MongoDB using Mongoose through a dedicated configuration file. The connectDB function is responsible for establishing a connection between the backend server and the MongoDB database.

The function is defined as an asynchronous function to handle the asynchronous nature of database connections. It uses mongoose.connect() and retrieves the MongoDB connection string from environment variables (process.env.MONGO_URI). Storing the database URI in a .env file improves security by preventing sensitive credentials from being hardcoded into the source code.

If the connection is successful, the application logs the host name of the connected MongoDB instance to the console. In case of a connection failure, an error message is displayed and the server process is terminated using process.exit(1). This ensures that the application does not continue running without a valid database connection.This configuration approach improves security, maintainability, and error handling while ensuring a stable connection between the backend and the MongoDB database.

4. Setup Instructions

This section explains the requirements and steps needed to set up and run the MERN Freelancing Platform on a local system.

Prerequisites

Before running the application locally, the following software must be installed on the system:

Node.js (LTS Version)
Node.js is required to run the backend server and manage project dependencies. It includes npm (Node Package Manager), which is used to install required packages for both frontend and backend. The recommended version is the latest LTS (Long Term Support) version.

Express.js
Express.js is used as the backend framework for building RESTful APIs. It is installed using npm as part of the backend dependencies.

MongoDB (Local Installation)
MongoDB must be installed and running locally on the system. The application connects to the local MongoDB server using the connection string:

mongodb://127.0.0.1:27017/mernapp

This creates or connects to a database named mernapp.

Code Editor (e.g., Visual Studio Code)
A code editor is required to modify and manage project files.

Web Browser (e.g., Google Chrome)
A modern web browser is required to access and test the frontend application.

Environment Variables Configuration

The backend application uses environment variables to securely store configuration details such as server port, database connection string, and authentication secret key. These values are defined inside a .env file located in the backend directory.

The following environment variables are used in this project:

```
PORT=5000
```

MONGO_URI=mongodb://127.0.0.1:27017/mernapp

JWT_SECRET=supersecretkey

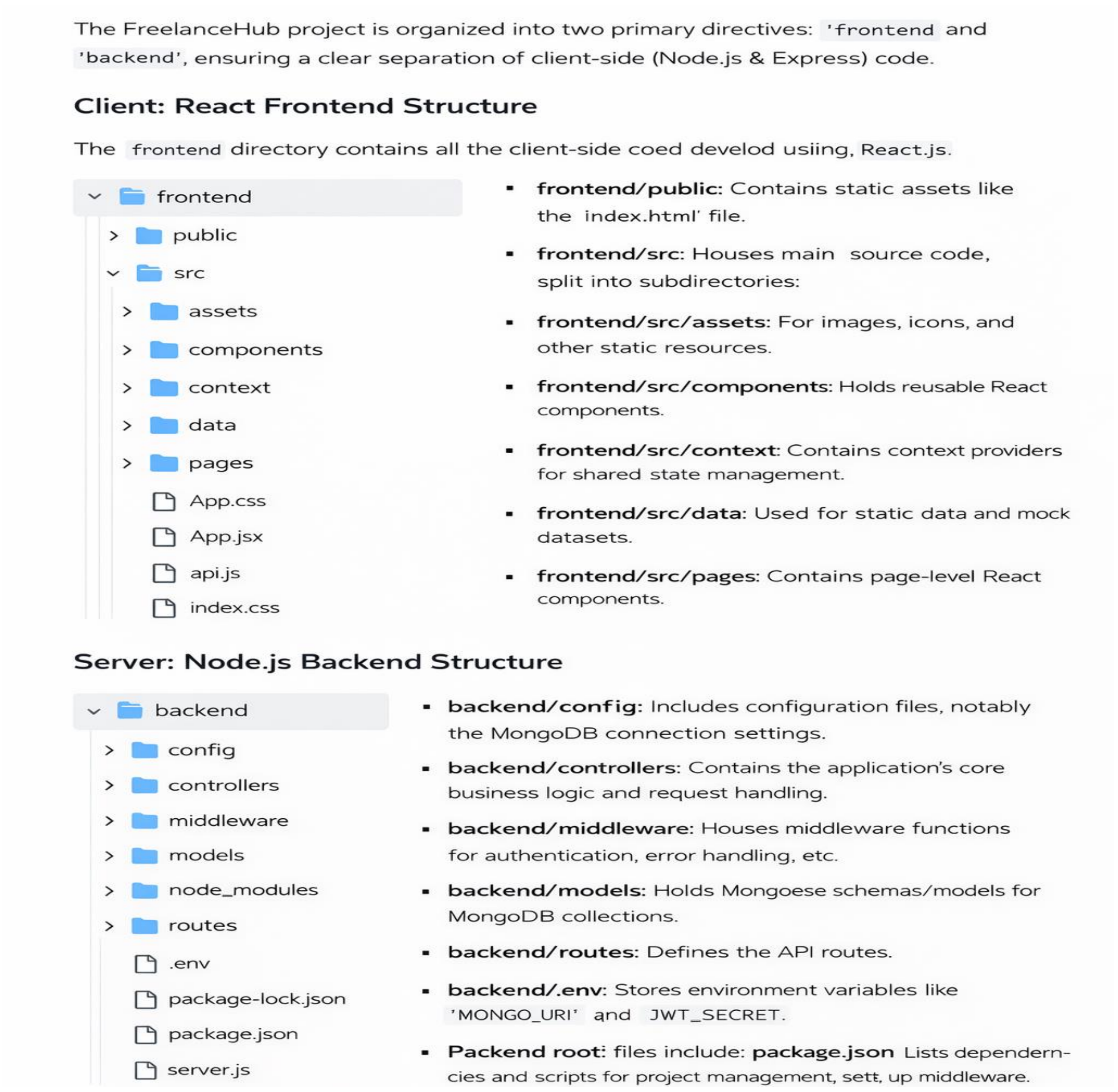
The PORT variable defines the port number on which the backend server runs. In this project, the server runs on port 5000.

The MONGO_URI variable specifies the MongoDB connection string. In this case, the application connects to a locally installed MongoDB instance running on 127.0.0.1 at port 27017, and uses a database named mernapp.

The JWT_SECRET variable is used to sign and verify JSON Web Tokens for authentication and authorization. This secret key ensures that tokens cannot be tampered with. In a production environment, this key should be strong, unique, and kept confidential.

Using environment variables enhances security by preventing sensitive information from being hardcoded into the application source code. It also allows easy configuration changes without modifying the core application logic.

5. Folder Structure



The FreelanceHub project is organized into two primary directives: `'frontend` and `'backend'`, ensuring a clear separation of client-side (Node.js & Express) code.

Client: React Frontend Structure

The `frontend` directory contains all the client-side coed develod usiing, React.js.

```
frontend
  public
  src
    assets
    components
    context
    data
    pages
    App.css
    App.jsx
    api.js
    index.css
```

- **frontend/public:** Contains static assets like the index.html' file.
- **frontend/src:** Houses main source code, split into subdirectories:
- **frontend/src/assets:** For images, icons, and other static resources.
- **frontend/src/components:** Holds reusable React components.
- **frontend/src/context:** Contains context providers for shared state management.
- **frontend/src/data:** Used for static data and mock datasets.
- **frontend/src/pages:** Contains page-level React components.

Server: Node.js Backend Structure

```
backend
  config
  controllers
  middleware
  models
  node_modules
  routes
  .env
  package-lock.json
  package.json
  server.js
```

- **backend/config:** Includes configuration files, notably the MongoDB connection settings.
- **backend/controllers:** Contains the application's core business logic and request handling.
- **backend/middleware:** Houses middleware functions for authentication, error handling, etc.
- **backend/models:** Holds Mongoese schemas/models for MongoDB collections.
- **backend/routes:** Defines the API routes.
- **backend/.env:** Stores environment variables like 'MONGO_URI' and JWT_SECRET.
- **Packend root:** files include: **package.json** Lists dependern-cies and scripts for project management, sett, up middleware.

Frontend (Client)

Inside the frontend folder:

- The public folder contains static files such as the main index.html file.
- The src folder contains the core React application code.
    - assets stores images, icons, and static resources.
    - components contains reusable React components such as Navbar, ProtectedRoute, etc.
    - context manages global state using React Context API.
    - data is used for static or mock data.
    - pages contains page-level components like Login, Register, Dashboard.
- App.jsx defines routing and main layout.
- main.jsx is the entry point that renders the React application.
- api.js handles communication with the backend server.
- App.css and index.css manage styling.

This structure ensures modular development, reusability of components, and organized routing.

Backend (Server)

Inside the backend folder:

- config contains configuration files such as database connection setup.
- controllers handle business logic and request processing.
- middleware contains authentication and error-handling logic.
- models defines MongoDB schemas using Mongoose.
- routes defines API endpoints and connects them to controllers.
- .env stores environment variables like MONGO_URI and JWT_SECRET.
- package.json manages dependencies.
- server.js is the main entry point of the backend application.

This backend structure follows the MVC (Model-View-Controller) pattern, which separates database logic, business logic, and routing for better maintainability.

6. Running the Application

This section describes how to start the frontend and backend servers locally after completing the installation and setup process.

Before running the application, ensure that MongoDB is running on your local system and that all dependencies have been installed in both the frontend and backend directories.

Starting the Backend Server

First, open a terminal and navigate to the backend directory:

cd backend

Then start the backend server using:

npm start

The server will start on the port specified in the .env file (default: PORT=5000).
If the database connection is successful, a confirmation message such as *"MongoDB Connected"* will appear in the terminal.

The backend API will be accessible at:

http://localhost:5000

Starting the Frontend Server

Open a new terminal window and navigate to the frontend directory:

cd frontend

Start the React development server using:

npm start

This will launch the frontend application in your default browser at:

http://localhost:3000

Application Execution Flow

Once both servers are running:

1. The React frontend runs on port 3000.
2. The Express backend runs on port 5000.
3. The frontend communicates with the backend through REST API calls.
4. The backend interacts with MongoDB to store and retrieve data.

The application is now fully functional in the local development environment.

7. API Documentation

The backend exposes RESTful API endpoints for handling authentication, user management, project operations, and bidding functionality. All responses are returned in JSON format. Protected routes require a valid JWT token in the request header.

Base URL:

http://localhost:5000/api

Authentication APIs

Register User

Endpoint

POST /api/auth/register

Description
Registers a new user (Client or Freelancer).

Request Body

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "123456",
  "role": "freelancer"
}
```

Response

```
{
  "message": "User registered successfully",
  "token": "jwt_token_here",
  "user": {
    "_id": "65abc123",
    "name": "John Doe",
    "email": "john@example.com",
    "role": "freelancer"
  }
}
```

Login User

Endpoint

POST /api/auth/login

Description
Authenticates a user and returns a JWT token.

Request Body

```
{
  "email": "john@example.com",
  "password": "123456"
}
```

Response

```
{
  "message": "Login successful",
  "token": "jwt_token_here",
  "user": {
    "_id": "65abc123",
    "name": "John Doe",
    "role": "freelancer"
  }
}
```

User APIs

Get User Profile

Endpoint

```
GET /api/users/profile
```

Headers

```
Authorization: Bearer <token>
```

Response

```
{
  "_id": "65abc123",
  "name": "John Doe",
  "email": "john@example.com",
  "role": "freelancer"
}
```

Project APIs

Create Project

Endpoint

```
POST /api/projects
```

Headers

```
Authorization: Bearer <token>
```

Request Body

```
{
  "title": "Website Development",
  "description": "Build a MERN website",
  "budget": 10000,
  "deadline": "2026-03-01"
}
```

Response

```
{
  "message": "Project created successfully",
  "project": {
    "_id": "789xyz",
    "title": "Website Development",
    "status": "Open"
  }
}
```

Get All Projects

Endpoint

GET /api/projects

Response

```
[
  {
    "_id": "789xyz",
    "title": "Website Development",
    "budget": 10000,
    "status": "Open"
  }
]
```

Update Project

Endpoint

PUT /api/projects/:id

Headers

Authorization: Bearer <token>

Response

```
{
  "message": "Project updated successfully"
}
```

Delete Project

Endpoint

DELETE /api/projects/:id

Headers

Authorization: Bearer <token>

Response

```
{
  "message": "Project deleted successfully"
}
```

Bid APIs

Submit Bid

Endpoint

POST /api/bids

Headers

Authorization: Bearer <token>

Request Body

```
{
  "projectId": "789xyz",
  "amount": 9000,
  "proposal": "I can complete this project in 10 days."
}
```

Response

```
{
  "message": "Bid submitted successfully"
}
```

Get Bids for a Project

Endpoint

GET /api/bids/:projectId

Response

```
[
  {
    "freelancer": "John Doe",
    "amount": 9000,
    "proposal": "I can complete this project in 10 days."
  }
]
```

Authentication & Security

Protected routes require the following header:

Authorization: Bearer <JWT_TOKEN>

The backend verifies the token using middleware before allowing access to secured endpoints.

8. Authentication and Authorization

The application implements secure authentication and authorization using **JSON Web Tokens (JWT)**. This ensures that only registered and authorized users can access protected resources within the system.

Authentication Process

Authentication is handled during user registration and login. When a user registers, their password is securely hashed using a cryptographic hashing algorithm before being stored in the MongoDB database. This prevents plain-text passwords from being stored and enhances data security.

During login, the user provides their email and password. The backend verifies the credentials by comparing the entered password with the hashed password stored in the database. If the credentials are valid, the server generates a JSON Web Token (JWT).

The generated JWT contains encoded user information such as the user ID and role. This token is signed using a secret key defined in the .env file (JWT_SECRET). The token is then sent back to the client.

Token-Based Authorization

After successful login, the frontend stores the JWT (commonly in localStorage). For every protected API request, the token is included in the request header as:

Authorization: Bearer <JWT_TOKEN>

On the backend, a custom authentication middleware verifies the token using the secret key. If the token is valid, the user is granted access to the requested resource. If the token is invalid or expired, access is denied and an error response is returned.

Role-Based Access Control

The system implements role-based authorization to restrict access to specific functionalities. Each user has a role (such as client or freelancer) stored in the database.

Middleware checks the user's role before allowing access to certain endpoints. For example:

- Only clients can create or delete projects.
- Only freelancers can submit bids.
- Protected dashboards are accessible only to authenticated users.

This ensures that users can perform only the actions permitted for their role.

Security Measures

The project follows several security best practices:

- Passwords are hashed before storage.
- Sensitive credentials such as the JWT secret and database URI are stored in environment variables.
- Protected routes use middleware for token verification.
- Unauthorized access attempts return appropriate error responses.

Session Management

The application uses stateless authentication, meaning no session data is stored on the server. Instead, authentication relies entirely on JWT tokens. This approach improves scalability and simplifies deployment since the server does not need to manage session storage. Overall, the authentication and authorization system ensures secure access control, protects sensitive data, and maintains a structured permission system within the application.

9. User Interface

The User Interface (UI) of the application is developed using **React.js** with a responsive and component-based design. The interface is designed to provide a clean layout, smooth navigation, and an intuitive experience for both clients and freelancers. The frontend dynamically updates based on API responses, ensuring real-time interaction with the backend.

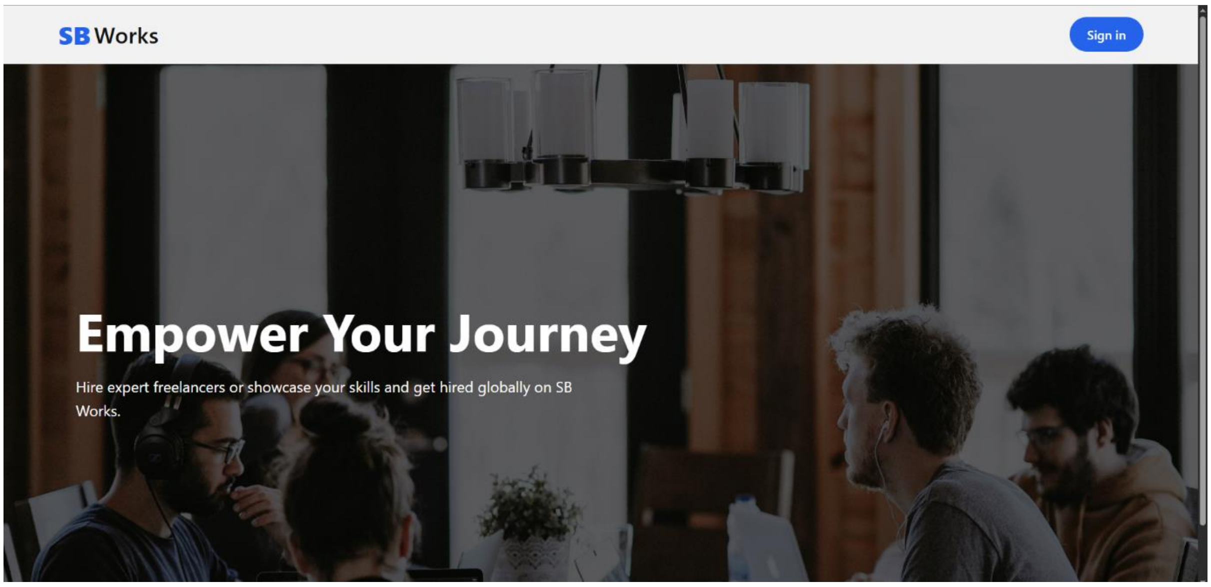
Below are key UI sections of the application:

The **landing page** serves as the entry point of the application. It contains a navigation bar, project introduction section, and call-to-action buttons such as Login and Register. The design is responsive and adapts to different screen sizes.

The **login page** allow users to securely access the platform. These pages include form validation, error handling messages, and secure input handling for credentials.

The **Client dashboard** shows the total number of projects that are in the progress and the total number of applications and the total number of projects which helps the clients to know how much of the work is being accepted and work in progress and **FreeLancer Dashboard** tells how much the he earned for his work and the other information

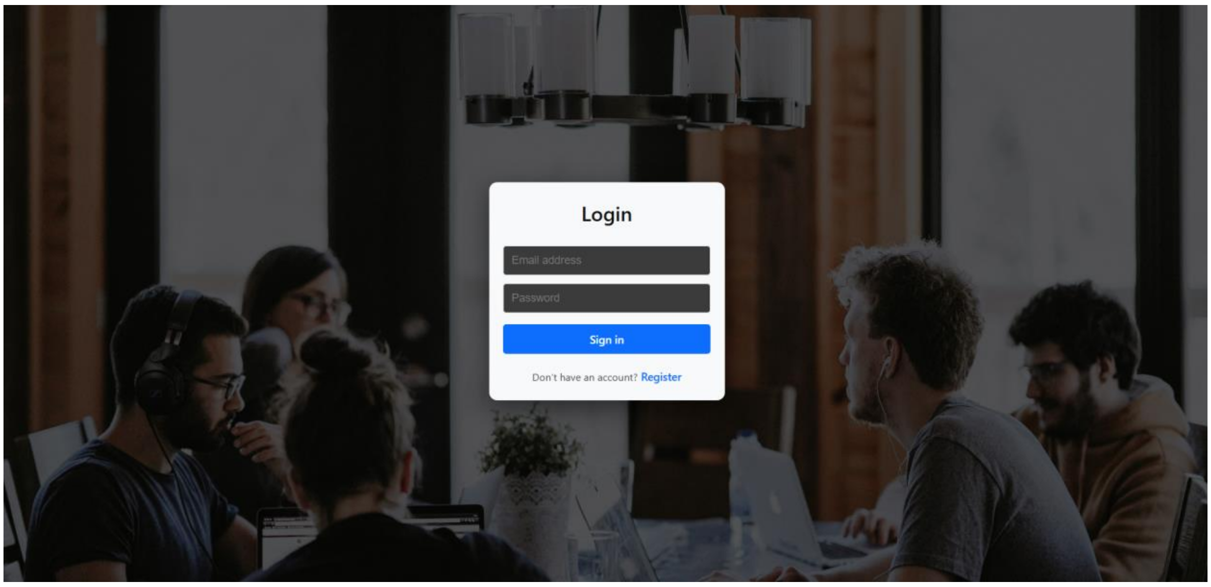
Landing Page



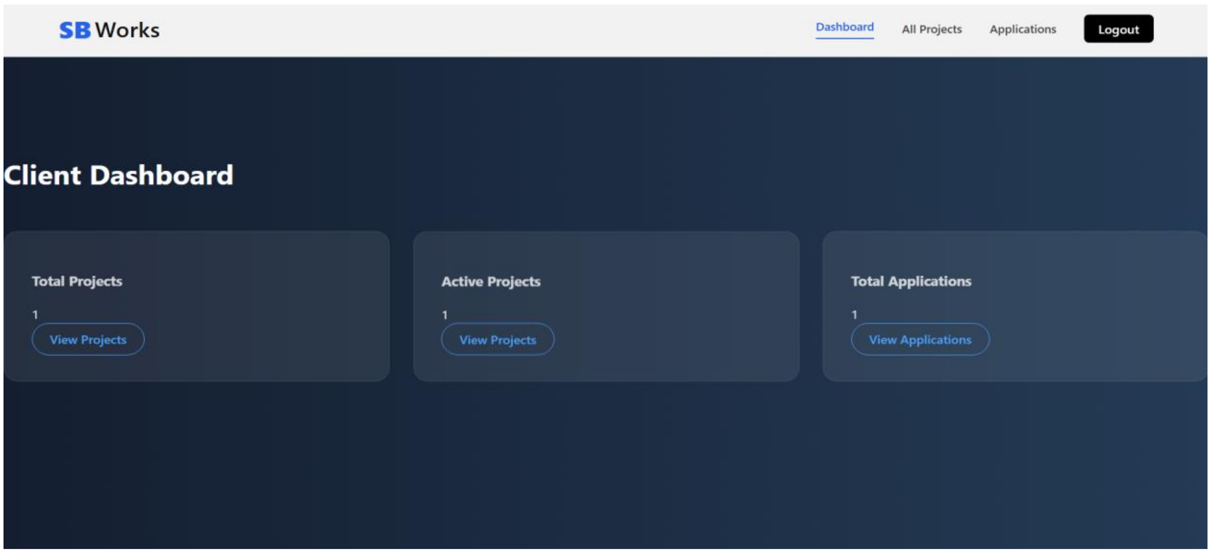
Empower Your Journey

Hire expert freelancers or showcase your skills and get hired globally on SB Works.

Login Page



Client Dashboard



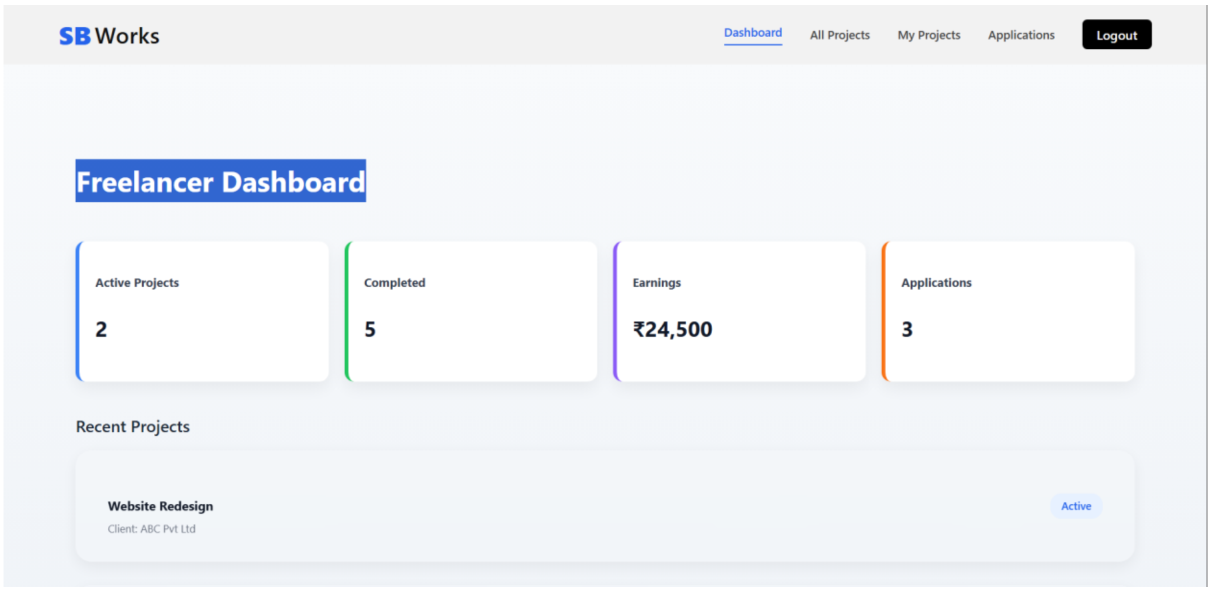
Client Dashboard

Total Projects
1
View Projects

Active Projects
1
View Projects

Total Applications
1
View Applications

Free Lancer Dashboard



Freelancer Dashboard

Active Projects
2

Completed
5

Earnings
₹24,500

Applications
3

Recent Projects

Website Redesign
Client: ABC Pvt Ltd
Active

10. Testing

Testing plays a crucial role in ensuring the reliability, security, and performance of the MERN Freelancing Platform. The application follows a structured testing strategy that includes manual testing, API testing, and basic validation testing to verify both frontend and backend functionalities.

Testing Strategy

The project follows a combination of functional testing and integration testing to ensure smooth communication between the frontend, backend, and database.

Functional testing is performed to verify that all features such as user registration, login, project posting, bidding, and dashboard operations work as expected. Each feature is tested individually to ensure correctness.

Integration testing ensures that the frontend correctly communicates with the backend APIs and that data is properly stored and retrieved from MongoDB.

Validation testing is applied on both frontend and backend to check for invalid inputs, missing fields, and incorrect data formats.

Authentication testing verifies that protected routes cannot be accessed without a valid JWT token and that role-based restrictions function correctly.

Tools Used for Testing

ManualTesting
The application is manually tested through the browser by interacting with all features such as form submissions, navigation, and dashboard updates.

ThunderClient
Postman is used to test backend API endpoints independently. It helps verify request methods (GET, POST, PUT, DELETE), headers, request bodies, and response structures.

MongoDBCompass
MongoDB Compass is used to inspect database collections and verify that data is correctly stored, updated, and deleted.

BrowserDeveloperTools
The browser's developer tools (Network tab and Console tab) are used to monitor API calls, detect frontend errors, and debug issues.

Test Cases Covered

- User registration with valid and invalid inputs
- Login with correct and incorrect credentials
- JWT token generation and verification
- Project creation, update, and deletion
- Bid submission and retrieval
- Access restriction for unauthorized users
- Proper error handling and validation messages

Result

All core functionalities of the application were tested successfully. The system handles user authentication securely, performs CRUD operations correctly, and maintains proper communication between frontend, backend, and database layers.The testing process ensures that the application is stable, secure, and ready for deployment in a real-world environment.

11. Screenshots or Demo

The G-Drive Link of the Demo Video is below:

Click Here for the Demo Video

12. Known Issues

Although the application is fully functional, there are a few known limitations and minor issues that users or developers should be aware of. The application currently runs in a local development environment, and production-level deployment configurations such as HTTPS, advanced security headers, and performance optimization have not yet been implemented. Token expiration handling may require improvement. If a JWT token expires, the user must manually log in again, as automatic token refresh functionality is not implemented.

Role-based authorization is implemented at the middleware level; however, additional fine-grained permission controls can be enhanced to improve security. Form validation is implemented on both frontend and backend, but advanced validation techniques and more descriptive error handling messages can be further improved. The system does not currently include advanced features such as real-time notifications, payment gateway integration, or chat functionality (if not implemented).

Scalability optimizations such as database indexing strategies, caching mechanisms, and load balancing have not been configured since the application is designed primarily for academic and demonstration purposes. There is no automated testing framework (such as Jest or Mocha) integrated into the project; testing has been performed manually and using API testing tools. Despite these minor limitations, the core functionalities of authentication, project management, and bidding operate as intended in the local development environment.

13. Future Enhancements

The FreelanceHub platform has been developed with a scalable architecture that allows future expansion and improvement. Although the current system provides core freelancing functionalities, several enhancements can be implemented to improve performance, usability, and real-world applicability.

One potential enhancement is the integration of a **real-time messaging system** using technologies such as Socket.io. This would allow direct communication between clients and freelancers within the platform, improving collaboration and reducing dependency on external communication tools.

Another improvement could be the addition of a **secure payment gateway integration**. Implementing online payment systems would enable clients to securely pay freelancers through the platform, making it more practical for real-world usage.

The system can also be enhanced by implementing **real-time notifications** for events such as bid submissions, project updates, and status changes. This would improve user engagement and provide instant updates.

An **admin panel** can be developed to monitor users, projects, and platform activity. This would help manage reports, handle disputes, and maintain overall system integrity.

Performance optimization techniques such as **database indexing, caching mechanisms, and API rate limiting** can be implemented to improve scalability and handle higher user traffic efficiently.

Security can be further strengthened by adding **email verification, password reset functionality, multi-factor authentication (MFA), and refresh token mechanisms** for improved token management.

The frontend can be improved with enhanced UI/UX features, advanced filtering and search capabilities for projects, and improved dashboard analytics with graphical representations of user activity.

Finally, the application can be deployed using cloud platforms such as AWS, Azure, or Render, with containerization using Docker for better scalability and maintainability in a production environment.

Overall, these future enhancements would transform the project from an academic implementation into a fully production-ready freelancing platform.