

Full Stack Development with MERN

Project Documentation format

1. Introduction

- Project Title: [Your Project Title]
- Team Members: List team members and their roles.

2. Project Overview

- Purpose: Briefly describe the purpose and goals of the project.
- Features: Highlight key features and functionalities.

3. Architecture

- Frontend: Describe the frontend architecture using React.
- Backend: Outline the backend architecture using Node.js and Express.js.
- Database: Detail the database schema and interactions with MongoDB.

4. Setup Instructions

- Prerequisites: List software dependencies (e.g., Node.js, MongoDB).
- Installation: Step-by-step guide to clone, install dependencies, and set up the environment variables.

5. Folder Structure

- Client: Describe the structure of the React frontend.
- Server: Explain the organization of the Node.js backend.

6. Running the Application

- Provide commands to start the frontend and backend servers locally.
 - o Frontend: npm start in the client directory.
 - o Backend: npm start in the server directory.

7. API Documentation

- Document all endpoints exposed by the backend.
- Include request methods, parameters, and example responses.

8. Authentication

- Explain how authentication and authorization are handled in the project.
- Include details about tokens, sessions, or any other methods used.

9. User Interface

- Provide screenshots or GIFs showcasing different UI features.

10. Testing

- Describe the testing strategy and tools used.

11. Screenshots or Demo

- Provide screenshots or a link to a demo to showcase the application.

12. Known Issues

- Document any known bugs or issues that users or developers should be aware of.

13. Future Enhancements

- Outline potential future features or improvements that could be made to the project.

Full Stack Development with MERN

Project Documentation

1. Introduction:

1.1 Project Title: SB Works – Freelancing Platform

1.2 Team:

Team Leader: Challa Charmila

Team Member: Mohammad Ansar Ali

Team Member: Akshitha Muthyala

Team Member: Kotturi Sai Sowmya

2. Project Overview:

2.1 Purpose of the Project:

SB Works is a web-based freelancing platform designed to connect clients and freelancers under administrative supervision. The primary objective of this project is to create a structured environment where clients can post projects, freelancers can apply and submit work, and administrators can monitor system activities and manage users.

The application supports three major roles: Admin, Client, and Freelancer. Each role has dedicated dashboards and functionalities. Clients are allowed to create and manage projects. Freelancers can browse available projects, apply to them, and submit completed work. Administrators have system-level visibility and can monitor users, projects, and applications.

The system also integrates real-time communication functionality using socket technology, allowing direct interaction between users.

The goal is to provide a structured platform for:

- Posting projects
- Applying to projects
- Managing applications
- Submitting work
- Reviewing freelancers
- Real-time communication (chat)

The platform enables clients to post projects, freelancers to apply and submit work, and administrators to monitor system activities. It also supports real-time chat functionality.

2.2 Core Features:

SB Works includes multiple features categorized by user roles.

Authentication and Security Features

The application implements secure user registration and login mechanisms using password hashing and JWT authentication. OTP verification is included to validate user identity.

Key authentication features include:

- User registration with role selection
- Secure login with JWT token generation
- OTP verification process
- Protected routes
- Role-based authorization middleware

Admin Features

The administrator dashboard provides complete oversight of platform operations.

- View all registered users
- Monitor projects created by clients
- Review freelancer applications
- Access system-level dashboard statistics

Client Features

Clients can manage projects from creation to completion.

- Create new projects with detailed descriptions
- View submitted applications from freelancers
- Select freelancers for projects
- Review submitted work
- Provide ratings or feedback

Freelancer Features

Freelancers can interact with projects and clients.

- Browse available projects
- Submit proposals for projects
- Track applied projects
- Submit completed work
- Communicate with clients through chat

3. Architecture:

The SB Works application follows a layered architecture separating frontend, backend, and database systems. The React frontend communicates with the Express backend via REST APIs. The backend interacts with MongoDB for persistent data storage.

The architecture ensures scalability, modularity, and maintainability. Communication between client and server occurs through HTTP requests secured with JWT tokens.

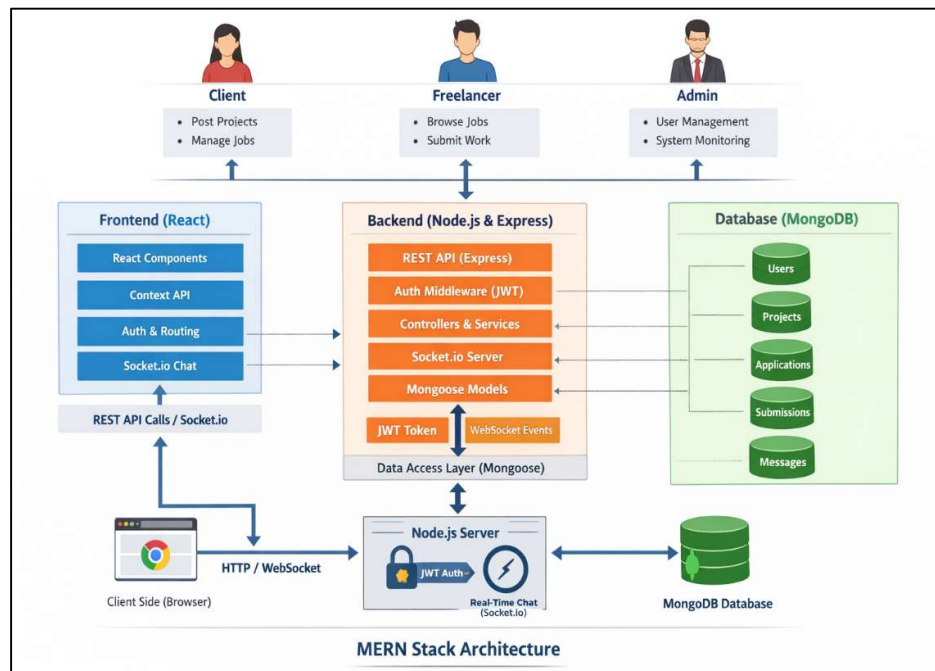


Fig 1: Project Architecture

3.1 Frontend Architecture – React Application Design

• Overview of the Frontend Architecture

The frontend of the project is developed using React and follows a Single Page Application (SPA) architecture. This design enables dynamic content rendering without requiring full page reloads, thereby improving user experience and performance. The frontend is responsible for handling user interactions, rendering views, managing application state, and communicating with the backend through RESTful APIs.

The architecture is modular and component-driven. Each part of the interface is divided into reusable components, ensuring scalability and maintainability. The frontend does not directly communicate with the database; instead, it interacts with the backend server through HTTP requests.

• Component-Based Design

The application UI is structured into reusable and independent React components. This promotes separation of concerns and prevents duplication of code.

Layout components such as the navigation bar, footer, and dashboard wrapper maintain consistency across pages. Page components represent complete views such as Login, Register, Dashboard, and entity management screens. Reusable components like form inputs, tables, buttons, modals, and alerts are used across multiple pages to ensure consistent styling and behavior.

This modular structure ensures that any updates or changes to UI logic can be handled efficiently without affecting unrelated parts of the application.

- **Routing Mechanism**

The frontend implements client-side routing using React Router. Routing enables smooth navigation between different views without refreshing the browser.

Public routes include pages like login and registration, which are accessible to unauthenticated users. Protected routes include dashboard and management pages, which are accessible only after successful authentication.

A custom route protection mechanism is implemented to verify authentication status before rendering restricted pages. If a valid authentication token is not found, users are redirected to the login page. This ensures proper access control at the UI level.

- **State Management**

State management in the frontend is handled primarily using React hooks such as `useState` and `useEffect`. These hooks allow components to manage local state and lifecycle events efficiently.

Authentication state, user role information, and application data retrieved from the backend are stored in component state or global context (if Context API is used). The state updates dynamically when API responses are received, triggering re-rendering of components.

This unidirectional data flow ensures predictable behavior and simplifies debugging and maintenance.

- **API Communication Layer**

The frontend communicates with the backend using HTTP requests through Axios or the Fetch API. All API calls are centralized in a service layer to improve maintainability.

For protected endpoints, the JSON Web Token (JWT) received during login is attached to the request header as an Authorization token. This ensures that only authenticated users can access restricted resources.

Error handling is managed by capturing API response errors and displaying appropriate feedback messages to users.

- **Authentication Handling in Frontend**

Authentication is implemented using token-based authentication. After a successful login, the backend returns a JWT token, which is stored securely in browser storage.

The token is included in subsequent requests to authenticate the user. Upon logout, the token is removed, and the application state is reset.

Conditional rendering is used to display different navigation menus and pages depending on whether the user is authenticated and based on their role.

3.2 Backend Architecture – Node.js and Express.js Design

- **Overview of the Backend Architecture**

The backend of the project is built using Node.js with Express.js as the web framework. It follows a layered architecture that separates routing, business logic, middleware processing, and database interactions.

The backend exposes RESTful APIs that serve as the communication bridge between the frontend and the database. It handles authentication, validation, business rules, and data persistence.

- **Server Structure and Organization**

The backend is organized into clearly defined directories to maintain modularity.

The main server file initializes the Express application and configures middleware, database connection, and routes. Route files define endpoint paths and connect them to controller functions. Controllers contain the business logic for handling requests and responses. Models define the structure of data stored in MongoDB. Middleware handles authentication, authorization, and error processing.

This separation ensures clean architecture and easier debugging.

- **Routing Architecture**

The routing layer defines API endpoints and maps them to corresponding controller functions. Each resource (such as users or project entities) has its own route file.

Routes handle HTTP methods such as GET, POST, PUT, and DELETE, ensuring RESTful design principles are followed. This structured routing approach improves scalability and clarity.

- **Controller Layer and Business Logic**

Controllers process incoming requests, validate input data, and interact with database models. They are responsible for implementing business rules and sending structured JSON responses back to the frontend.

Controllers do not directly handle authentication logic; instead, authentication is managed through middleware functions. This separation ensures better organization of code responsibilities.

- **Middleware Implementation**

Middleware plays a critical role in request processing. Authentication middleware verifies the JWT token included in incoming requests. Authorization middleware ensures that only users with appropriate roles can access certain endpoints.

Error-handling middleware captures exceptions and returns consistent error responses. Validation middleware ensures that incoming data meets defined schema requirements before processing.

This layered processing enhances security and maintainability.

- **Authentication Mechanism**

Authentication is implemented using JSON Web Tokens. During login, the user's credentials are verified, and a signed token is generated using a secret key. The token includes encoded user information such as user ID and role.

For protected routes, middleware verifies the token before allowing access to controller functions. Since JWT is stateless, the server does not store session information, making the application scalable.

3.3 Database Architecture – MongoDB and Data Modeling

- **Overview of Database Design**

The project uses MongoDB as its primary database. MongoDB is a NoSQL document-based database that stores data in JSON-like format (BSON).

The database structure is flexible, allowing the application to evolve without rigid schema constraints. Mongoose is used as an Object Data Modeling (ODM) library to define schemas and manage database operations.

- **Schema Design and Collections**

Each major entity in the application corresponds to a MongoDB collection. The primary collection is the Users collection, which stores authentication and role information.

The User schema includes fields such as name, email, password (hashed), role, and timestamps. Passwords are stored securely after hashing.

The application also includes domain-specific collections such as projects, records, or other business entities. These collections typically include fields like title, description, status, and a reference to the user who created them.

- **Relationships and References**

Although MongoDB is a NoSQL database, relationships are maintained using ObjectId references. For example, a project record may contain a reference field linking it to the User collection.

Mongoose provides a `populate()` method that allows related documents to be retrieved together, simulating relational database joins while maintaining NoSQL flexibility.

- **Database Operations**

All CRUD operations are performed using Mongoose model methods. Data creation, retrieval, updating, and deletion are handled using asynchronous functions.

Validation rules are defined at the schema level to enforce required fields, uniqueness constraints, and data type validation. This ensures data integrity before storing documents.

- **Interaction Flow Between Backend and Database**

When a request is received from the frontend, it passes through middleware and reaches the controller. The controller interacts with the Mongoose model, which communicates with MongoDB. The database returns the requested data, which is then formatted into a JSON response and sent back to the frontend.

This layered interaction prevents direct exposure of the database and ensures security.

4. Setup Instructions

4.1 Prerequisites

Before setting up and running the project, certain software tools and dependencies must be installed on the system. These tools ensure that both the frontend and backend environments function correctly.

The application is built using the MERN stack; therefore, it requires a JavaScript runtime environment, a package manager, and a database server.

The primary prerequisite is **Node.js**, which provides the runtime environment for executing JavaScript on the server side. Node.js also includes npm (Node Package Manager), which is used to install project dependencies. It is recommended to install the latest LTS (Long Term Support) version to ensure stability.

The backend of the project relies on **MongoDB** as its database. MongoDB can be installed locally on the system or accessed through a cloud service such as MongoDB Atlas. If running locally, the MongoDB server must be installed and actively running before starting the backend server.

In addition to Node.js and MongoDB, a code editor such as Visual Studio Code is recommended for managing and editing project files. Git is also required to clone the project repository from version control.

The complete list of prerequisites is as follows:

- Node.js (LTS version recommended)
- npm (comes bundled with Node.js)
- MongoDB (Local installation or MongoDB Atlas account)
- Git (for cloning the repository)
- Code editor (e.g., VS Code)
- Web browser (e.g., Chrome, Edge, Firefox)

Optional but recommended tools include:

- Postman (for API testing)
- MongoDB Compass (for visual database management)

Ensuring these prerequisites are installed and properly configured will prevent runtime and dependency-related issues.

4.2 Installation Process

The installation process involves cloning the project, installing dependencies for both frontend and backend, configuring environment variables, and starting the servers.

Step 1: Clone the Repository

The first step is to obtain the project files from the repository. Using Git, the repository can be cloned into the local system.

Open a terminal or command prompt and execute the following command:

- `git clone`

After cloning, navigate into the project directory:

- `cd`

This project contains separate folders for the frontend (client) and backend (server).

Step 2: Install Backend Dependencies

Navigate to the server directory:

- `cd server`

Install all required backend dependencies using npm:

- `npm install`

This command reads the package.json file and installs all necessary libraries such as:

- express
- mongoose
- bcrypt
- jsonwebtoken
- cors
- dotenv
- nodemon (if included for development)

Once installation is complete, the backend environment will be ready for configuration.

Step 3: Configure Environment Variables

The backend requires certain environment variables for secure configuration. These variables are stored in a .env file inside the server directory.

Create a new file named:

.env

Inside the .env file, define the required variables such as:

PORT=5000

MONGO_URI=your_mongodb_connection_string

JWT_SECRET=your_secret_key

PORT specifies the server port number.

MONGO_URI contains the connection string to the MongoDB database.

JWT_SECRET is used for signing authentication tokens.

If using MongoDB Atlas, the connection string can be obtained from the Atlas dashboard. If using local MongoDB, the connection string will typically be:

mongodb://localhost:27017/<database_name>

Environment variables ensure sensitive information is not hardcoded into the source code.

Step 4: Install Frontend Dependencies

After setting up the backend, navigate to the client directory from the root folder:

- `cd client`

Install frontend dependencies using:

- `npm install`

This installs libraries such as:

- react
- react-router-dom
- axios
- other UI libraries (if included)

The frontend dependencies are defined in the client's package.json file.

Step 5: Running the Application

Once both frontend and backend dependencies are installed and environment variables are configured, the application can be started.

To start the backend server:

Navigate to the server directory:

- cd server

Then run:

- npm start

or (if nodemon is configured):

- npm run dev

The backend server should start on the specified port (e.g., <http://localhost:5000>).

To start the frontend application:

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

- cd client

Then run:

- npm start

This will launch the React development server, usually on:

<http://localhost:3000>

The frontend will automatically connect to the backend server if the API base URL is correctly configured.

Verification of Setup

After both servers are running:

- Open the browser at <http://localhost:3000>
- Ensure the homepage loads successfully
- Test user registration and login
- Verify that data is being stored in MongoDB

If both frontend and backend are functioning correctly and database operations are successful, the setup has been completed successfully.

Common Setup Issues and Troubleshooting

During installation, the following issues may occur:

- MongoDB connection errors – Ensure MongoDB service is running.
- Port already in use – Change PORT value in .env file.
- Missing dependencies – Re-run npm install.
- CORS errors – Ensure backend has CORS middleware enabled.

Proper verification of each setup step ensures smooth execution of the application.

5. Folder Structure

5.1 Client: Structure of the React Frontend

Overview of the Client-Side Structure

The client-side of the project is developed using React and follows a modular, component-based architecture. The structure is organized to separate concerns such as UI components, routing logic, API communication, and state management. This organization improves maintainability, scalability, and readability of the codebase.

The main client folder contains the source code, dependency configuration, and public assets required to render the application in the browser.

Root-Level Files in the Client Directory

At the root level of the client folder, several important configuration files are present.

The package.json file defines the project metadata, dependencies, and available npm scripts. It lists libraries such as react, react-router-dom, axios, and other UI-related dependencies.

The package-lock.json file ensures consistent dependency versions across installations.

The public folder contains static files such as the main HTML template (index.html), favicon, and other publicly accessible assets.

The src folder contains the main application source code and is the most important directory in the frontend structure.

Source Folder (src) Organization

The src directory is structured logically to divide responsibilities across different parts of the application.

The index.js file serves as the entry point of the React application. It renders the root component (App.js) into the DOM.

The App.js file acts as the central component that manages routing and overall application structure. It defines navigation routes and wraps protected routes where necessary.

Components Directory

The components folder contains reusable UI elements. These components are designed to be modular and reusable across multiple pages.

Examples of components typically include:

- Navigation Bar
- Footer
- Sidebar
- Form Input Components
- Button Components
- Modal Components
- Table Components
- Alert/Notification Components

Each component is responsible for rendering a specific part of the interface. This approach promotes reusability and consistency throughout the application.

Pages Directory

The pages folder contains higher-level components representing complete views or screens in the application.

Typical pages include:

- Login Page
- Registration Page
- Dashboard Page
- Entity Management Page
- Admin Panel Page
- Profile Page

Each page component combines multiple smaller components and handles API calls related to that particular view.

Services or API Directory

The services (or api) folder centralizes all API calls. This layer abstracts HTTP request logic away from UI components.

For example, authentication-related API calls are handled in a separate file, while entity-related API calls are grouped together.

This ensures:

- Cleaner components
- Easier debugging
- Better maintainability
- Centralized error handling

Context or State Management Directory (If Implemented)

If the project uses Context API or any global state management system, a context folder is included.

This directory manages:

- Authentication state
- User role
- Global data
- Shared application state

This structure prevents prop drilling and ensures efficient state sharing across components.

Assets and Styling

The client folder may also include:

- CSS files
- Module-based CSS
- Images
- Icons

Styling may be managed through global stylesheets, component-level styles, or a UI framework if included.

5.2 Server: Organization of the Node.js Backend

Overview of the Backend Structure

The backend of the project is built using Node.js and Express.js. It follows a modular layered architecture that separates routing, controllers, middleware, and data models. This separation ensures clean code organization and improves scalability.

The server folder acts as the core of backend logic and handles all communication between the frontend and the database.

Root-Level Files in the Server Directory

At the root of the server folder, key configuration files are present.

The `package.json` file defines backend dependencies such as `express`, `mongoose`, `bcrypt`, `jsonwebtoken`, `cors`, and `dotenv`.

The main entry file (`server.js` or `app.js`) initializes the Express application, connects to the database, configures middleware, and registers route files.

The `.env` file stores environment variables such as database connection strings and secret keys.

Configuration Directory

If included, the `config` folder handles database connection setup.

It typically contains:

- Database connection file
- Environment configuration

This file establishes a connection to MongoDB using Mongoose and ensures proper error handling during startup.

Models Directory

The `models` folder defines Mongoose schemas and data structures for MongoDB collections.

Each file inside this folder represents a specific entity in the database, such as:

- User model
- Project/Record model
- Other domain-specific models

Schemas define:

- Field types
- Required fields
- Default values
- Validation rules
- References to other collections

Models act as an interface between the application and MongoDB.

Routes Directory

The routes folder defines API endpoints. Each route file corresponds to a particular resource.

For example:

- authRoutes.js
- userRoutes.js
- projectRoutes.js

Routes specify HTTP methods (GET, POST, PUT, DELETE) and link them to corresponding controller functions.

This separation keeps routing logic clean and organized.

Controllers Directory

Controllers contain the core business logic of the application.

Each controller function:

- Receives request data
- Validates input
- Interacts with database models
- Sends structured JSON responses

Controllers do not define routes themselves but are called by route definitions.

Middleware Directory

The middleware folder contains functions that execute before reaching controller logic.

Common middleware includes:

- Authentication middleware (JWT verification)
- Authorization middleware (role-based access control)
- Error-handling middleware
- Request validation middleware

Middleware enhances security and enforces consistent request handling.

Database Interaction Layer

The backend uses Mongoose to interact with MongoDB. All CRUD operations are handled through model methods.

Database operations are asynchronous and use async/await syntax to handle promises effectively.

This ensures efficient request processing and prevents blocking of the Node.js event loop.

Backend Request Lifecycle

The backend follows a structured request lifecycle:

1. Client sends HTTP request.
2. Express receives the request.
3. Middleware processes authentication and validation.
4. Route forwards request to controller.
5. Controller interacts with model.
6. Model communicates with MongoDB.
7. Response is returned to client in JSON format.

This layered flow ensures security and clean separation of responsibilities.

6. Running the Application

To run the application successfully without any errors, we need to run both backend and frontend at the same time, both should be in the running state simultaneously. To run these both, we need the commands, the commands are:

For Backend : First navigate into the folder using the command “ cd folder_name(backend)

Next, run the command “ node index.js “.

For Frontend : First navigate into the folder using the command “ cd folder_name(frontend)

Next, run the command “ npm run dev “.

7. API Documentation

Overview of the API Design

The backend exposes a set of RESTful API endpoints that allow the frontend to perform authentication, user management, and domain-specific CRUD operations. All endpoints follow REST principles and return responses in JSON format.

The base URL for the API in local development is:

http://localhost:5000/api

All protected routes require an Authorization header containing a valid JWT token in the format:

Authorization: Bearer <token>

7.1. Authentication Endpoints

7.1.a. User Registration

Endpoint

POST /api/auth/register

Description

This endpoint allows new users to create an account in the system.

Request Body Parameters

- name (String, required) – Full name of the user
- email (String, required) – Unique email address
- password (String, required) – User password
- role (String, required) – User role

Example Request (JSON)

```
{  
  "name": "John Doe",  
  "email": "john@example.com",  
  "password": "Password123",  
  "role": "user"  
}
```

Success Response (201 Created)

```
{  
  "message": "User registered successfully",  
  "user": {  
    "_id": "65f1a2b3c4d5e6f7g8h9",  
    "name": "John Doe",  
    "email": "john@example.com",  
    "role": "user"  
  }  
}
```

Error Responses

400 Bad Request – Missing required fields

409 Conflict – Email already exists

7.1.b. User Login

Endpoint

POST /api/auth/login

Description

Authenticates a user and returns a JWT token.

Request Body Parameters

- email (String, required)
- password (String, required)

Example Request

```
{  
  "email": "john@example.com",  
  "password": "Password123"  
}
```

Success Response (200 OK)

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "_id": "65f1a2b3c4d5e6f7g8h9",
    "name": "John Doe",
    "role": "user"
  }
}
```

Error Responses

401 Unauthorized – Invalid credentials

404 Not Found – User does not exist

7.2. User Management Endpoints

7.2.a. Get All Users (Admin Only)

Endpoint

GET /api/users

Description

Returns a list of all registered users. Accessible only to Admin.

Headers

Authorization: Bearer <token>

Success Response

```
{
  "users": [
    {
      "_id": "123",
      "name": "John Doe",
      "email": "john@example.com",
      "role": "user"
    },
    {
      "_id": "456",
      "name": "Admin User",
      "email": "admin@example.com",
      "role": "admin"
    }
  ]
}
```

Error Response

403 Forbidden – Insufficient permissions

7.2.b. Get Single User

Endpoint

GET /api/users/:id

Description

Retrieves details of a specific user by ID.

URL Parameter

- id (String, required) – User ID

Success Response

```
{
  "_id": "123",
  "name": "John Doe",
  "email": "john@example.com",
  "role": "user"
}
```

7.2.c. Update User**Endpoint**

PUT /api/users/:id

Description

Updates user information.

Request Body

- name (optional)
- email (optional)
- role (admin only)

Example Request

```
{
  "name": "John Updated"
}
```

Success Response

```
{
  "message": "User updated successfully",
  "user": {
    "_id": "123",
    "name": "John Updated",
    "email": "john@example.com"
  }
}
```

7.2.d. Delete User (Admin Only)

Endpoint

DELETE /api/users/:id

Description

Deletes a user account.

Success Response

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

7.3. Domain Entity Endpoints (Projects / Records / Tasks)

(Note: Replace "projects" with your actual entity name if different.)

7.3.a. Create Record

Endpoint

POST /api/projects

Description

Creates a new project/record associated with the authenticated user.

Headers

Authorization: Bearer <token>

Request Body

- title (String, required)
- description (String, optional)
- status (String, optional)

Example Request

```
{  
  "title": "New Project",  
  "description": "Project description",  
  "status": "Active"  
}
```

Success Response

```
{  
  "message": "Project created successfully",  
  "project": {  
    "_id": "789",  
    "title": "New Project",  
    "status": "Active",  
    "createdBy": "123"  
  }  
}
```

```
}  
}  
}
```

7.3.b. Get All Records

Endpoint

GET /api/projects

Description

Retrieves all records for the authenticated user.

Success Response

```
{  
  "projects": [  
    {  
      "id": "789",  
      "title": "New Project",  
      "status": "Active"  
    }  
  ]  
}
```

7.3.c. Get Single Record

Endpoint

GET /api/projects/:id

Description

Retrieves a specific project by ID.

Success Response

```
{  
  "id": "789",  
  "title": "New Project",  
  "description": "Project description",  
  "status": "Active"  
}
```

7.3.d. Update Record

Endpoint

PUT /api/projects/:id

Description

Updates an existing project.

Request Body Example

```
{  
  "status": "Completed"  
}
```

Success Response

```
{
  "message": "Project updated successfully",
  "project": {
    "_id": "789",
    "status": "Completed"
  }
}
```

7.3.e. Delete Record

Endpoint

DELETE /api/projects/:id

Description

Deletes a specific project.

Success Response

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

7.4. Authorization and Security Requirements

Protected endpoints require:

- Valid JWT token
- Proper role authorization (for admin routes)

If token is missing:

401 Unauthorized

If token is invalid:

403 Forbidden

7.5. Standard API Response Format

Most responses follow a consistent structure:

For Success:

```
{
  "message": "Operation successful",
  "data": { ... }
}
```

For Errors:

```
{
  "error": "Error description"
}
```

7.6. HTTP Status Codes Used

200 – OK
201 – Created
400 – Bad Request
401 – Unauthorized
403 – Forbidden
404 – Not Found
500 – Internal Server Error

7.7. Complete API Flow Example

Login Flow:

Client → POST /api/auth/login → Server verifies → Returns JWT → Client stores token → Token sent in Authorization header for future requests.

CRUD Flow:

Client → Protected Route → Middleware verifies token → Controller executes → Mongoose interacts with MongoDB → JSON response returned.

8. Authentication

Overview of Security Mechanism

Authentication and authorization in this project are implemented using a token-based security model built on JSON Web Tokens (JWT). The system ensures that only verified users can access protected resources and that user permissions are enforced based on assigned roles.

The authentication process verifies the identity of users, while authorization determines what actions an authenticated user is allowed to perform within the system.

The application follows a stateless authentication approach, meaning the server does not maintain session data. Instead, authentication state is maintained using cryptographically signed tokens.

Authentication Process

- **User Registration**

During registration, the user provides required credentials such as name, email, and password. The password is not stored in plain text. Instead, it is hashed using a secure hashing algorithm (such as bcrypt) before being stored in the database.

This ensures that even if database data is exposed, raw passwords remain protected.

- **User Login**

When a user attempts to log in, the following steps occur:

1. The user submits email and password credentials.
2. The backend retrieves the user record from the database.
3. The submitted password is compared with the hashed password stored in the database.
4. If the credentials match, a JWT token is generated.
5. The token is sent back to the client.

If credentials are invalid, an authentication error response is returned.

- **JSON Web Token (JWT) Implementation**

Token Structure

The JWT token contains encoded information that includes:

- User ID
- User role
- Token expiration time

The token is digitally signed using a secret key stored in environment variables. This ensures that the token cannot be tampered with.

The signature guarantees data integrity and authenticity.

- **Token Storage and Usage**

After successful login, the token is sent to the frontend and stored in browser storage (typically localStorage or sessionStorage).

For every protected API request, the frontend attaches the token in the HTTP Authorization header using the format: Authorization: Bearer

The backend middleware extracts and verifies the token before granting access to protected routes.

- **Authentication Middleware**

Authentication middleware is implemented on the backend to secure routes.

When a protected endpoint is accessed:

1. The middleware checks for the presence of the Authorization header.
2. It extracts the token.
3. It verifies the token using the secret key.
4. If valid, it decodes user information and attaches it to the request object.
5. The request proceeds to the controller.

If the token is missing, invalid, or expired, access is denied with an appropriate error response.

This ensures that only authenticated users can access restricted endpoints.

- **Token Expiration and Security**

Tokens are configured with an expiration time to enhance security. After expiration, the user must log in again to obtain a new token.

This prevents long-term misuse of compromised tokens and enhances overall security.

- **Authorization Mechanism**

Authorization determines what actions a user can perform based on their role.

Each user has a role field stored in the database, such as:

- Admin
- User

After token verification, the decoded user role is checked by authorization middleware to determine whether the user has sufficient privileges to access a specific route.

- **Role-Based Access Control (RBAC)**

Role-Based Access Control is implemented to restrict certain actions.

For example:

- Admin users can manage all users and system data.
- Regular users can only manage their own records.
- Guests cannot access protected routes.

Authorization middleware checks the role value before allowing access to admin-specific endpoints.

If a user attempts to access a restricted resource without proper permissions, a 403 Forbidden response is returned.

- **Stateless Authentication Design**

The system uses a stateless authentication model. Unlike session-based authentication where session data is stored on the server, JWT-based authentication does not require the server to maintain session records.

Advantages of stateless authentication:

- Improved scalability
- Reduced server memory usage
- Easier horizontal scaling
- Suitable for distributed systems

Because the token carries all required authentication data, each request is independently verified.

- **Session Handling**

The project does not rely on traditional server-side sessions. Instead, the authentication state is maintained through JWT tokens.

Logout functionality is handled by:

- Removing the token from browser storage.
- Resetting authentication state in the frontend.

Since the server does not store session data, logging out simply invalidates the token on the client side.

- **Security Best Practices Implemented**

The following security practices are implemented in the authentication system:

- Password hashing using bcrypt
- Secure JWT signing using environment secret keys
- Token expiration to limit misuse
- Protected routes using authentication middleware
- Role-based authorization checks
- Environment variables for sensitive data
- CORS configuration to control cross-origin access

These measures ensure that user credentials and protected resources remain secure.

- **Complete Authentication Flow**

The overall authentication flow in the system is as follows:

1. User registers → password hashed → stored in database.
2. User logs in → credentials verified → JWT generated.
3. Token sent to client → stored securely.
4. Client sends token with protected requests.
5. Backend middleware verifies token.
6. Authorization middleware checks user role.
7. Access granted or denied accordingly.

9. User Interface

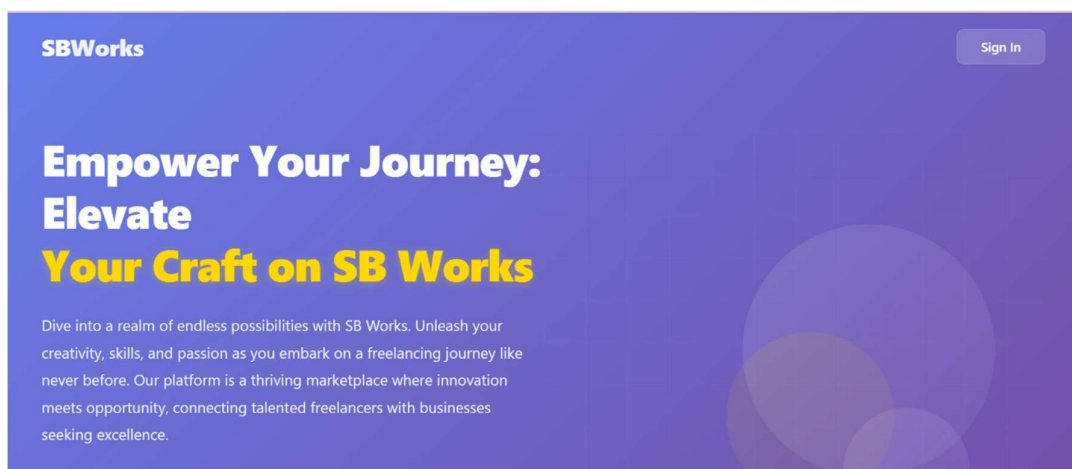


Fig 2: Website entry page

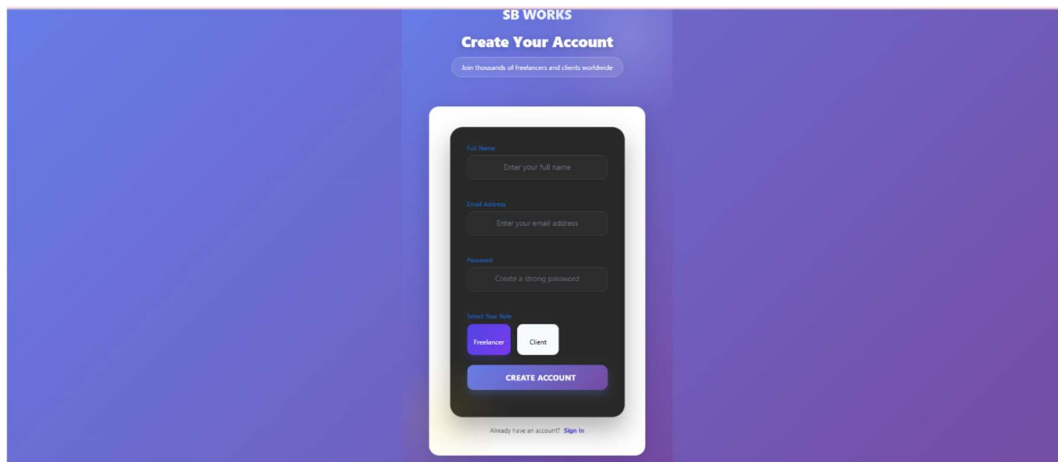


Fig 3: Register page

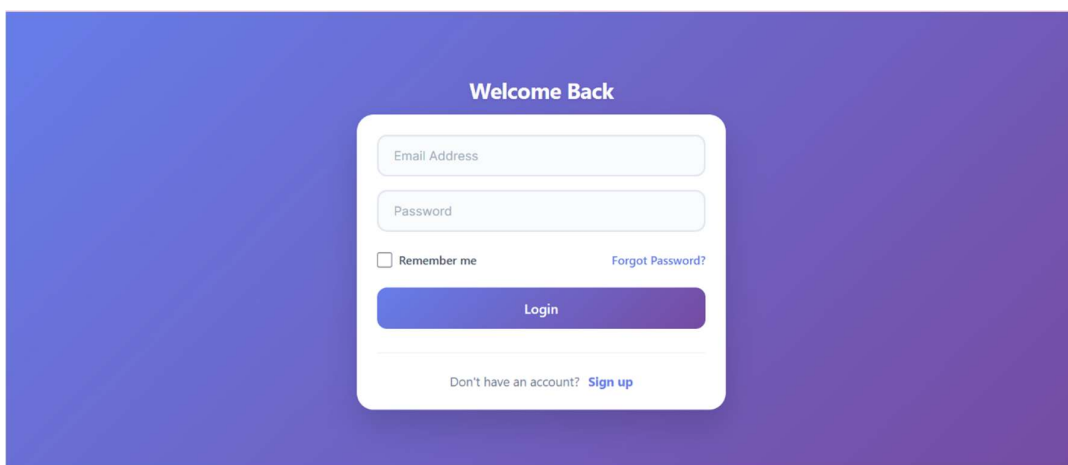


Fig 4: Logging Page

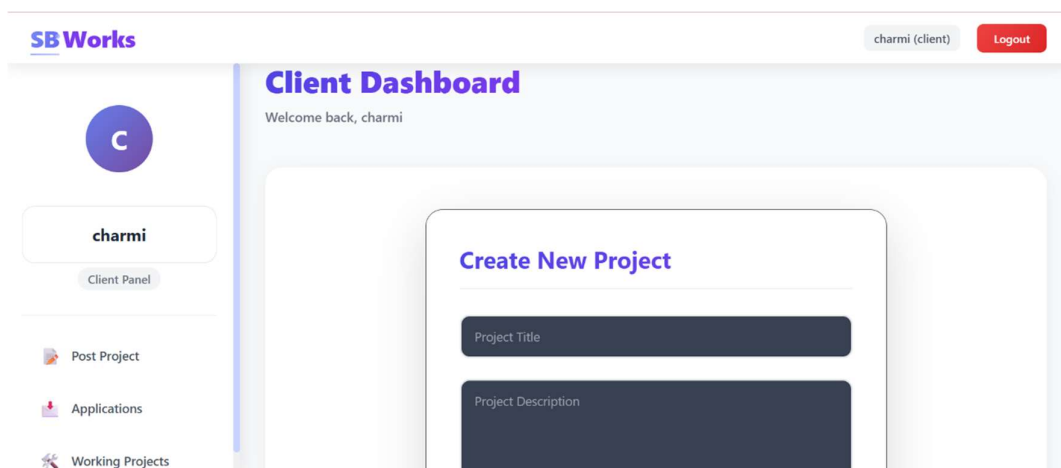


Fig 5: Client Dashboard

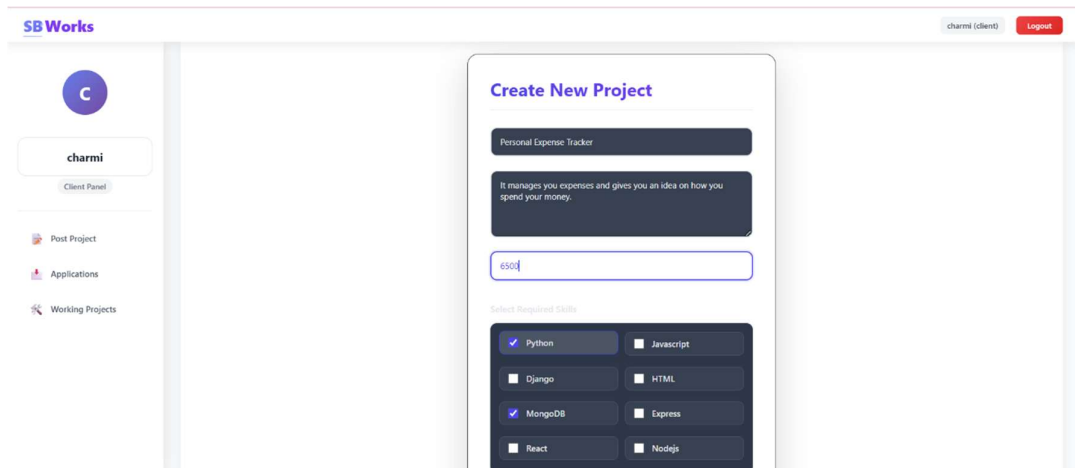


Fig 6: Uploading of project in client dashboard

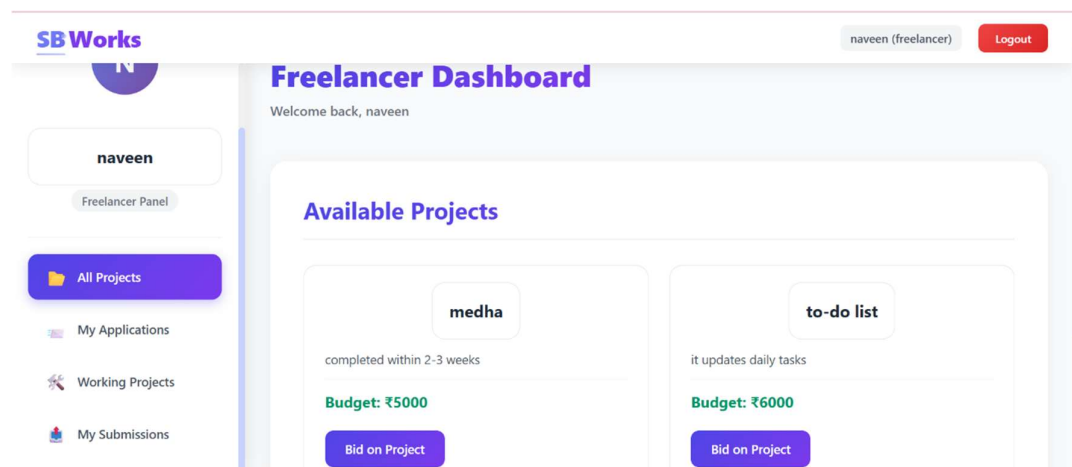


Fig 7: Freelancers shown all the projects from clients

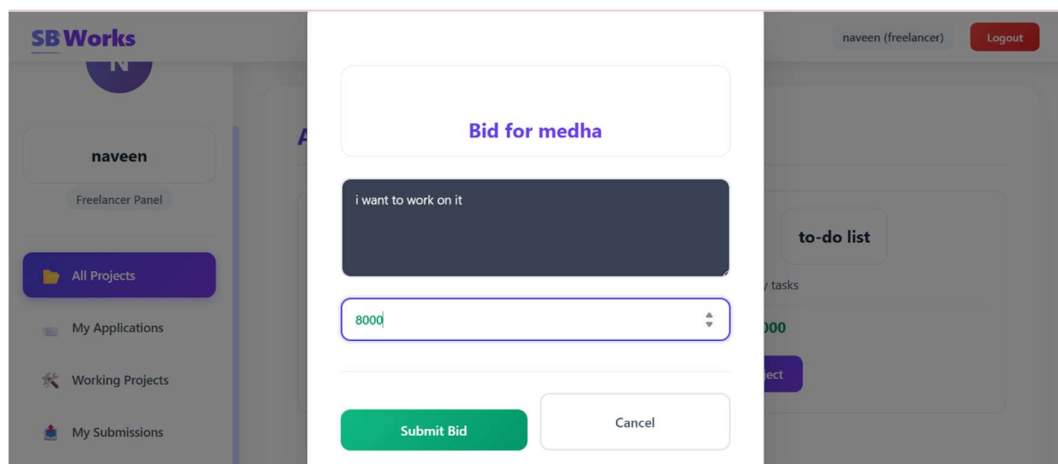


Fig 8: Freelancer bidding on the project

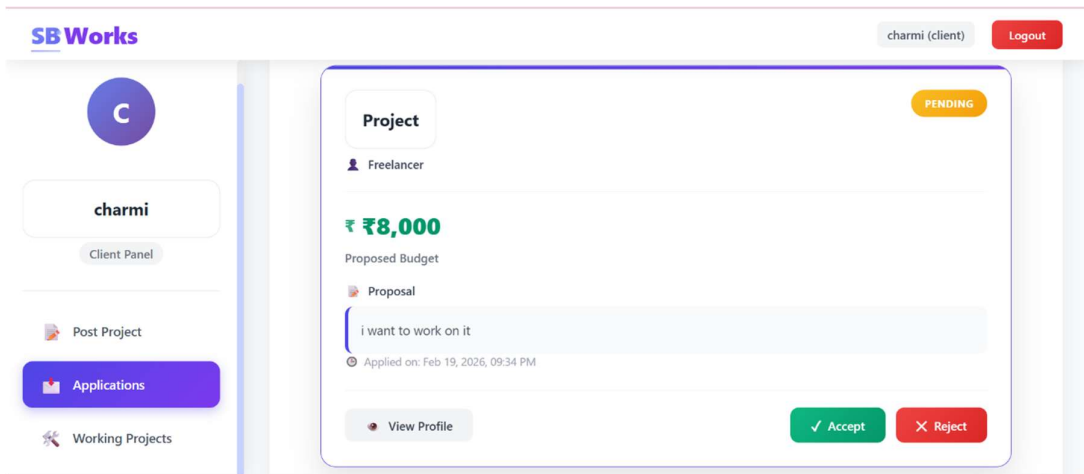


Fig 9: Client getting the bid from the freelancer

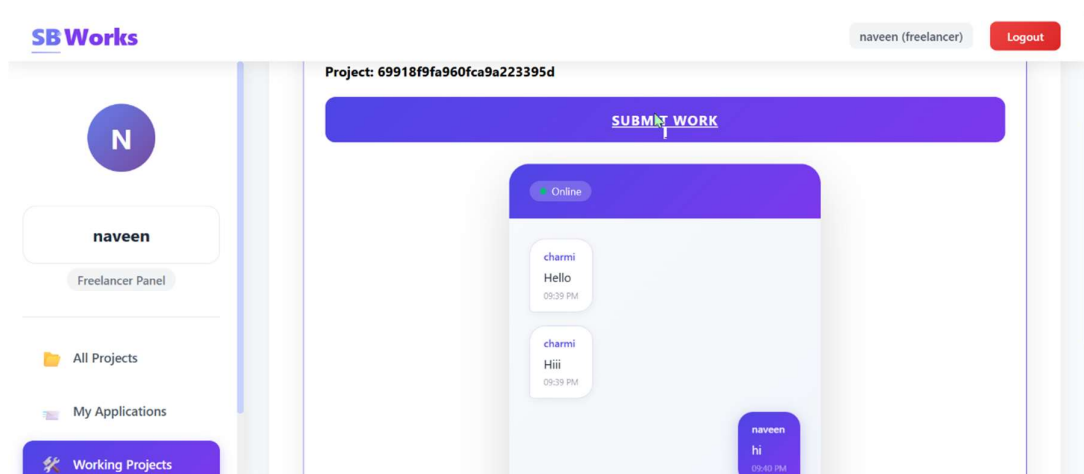


Fig 10: Conversation build up with the accepted Freelancer

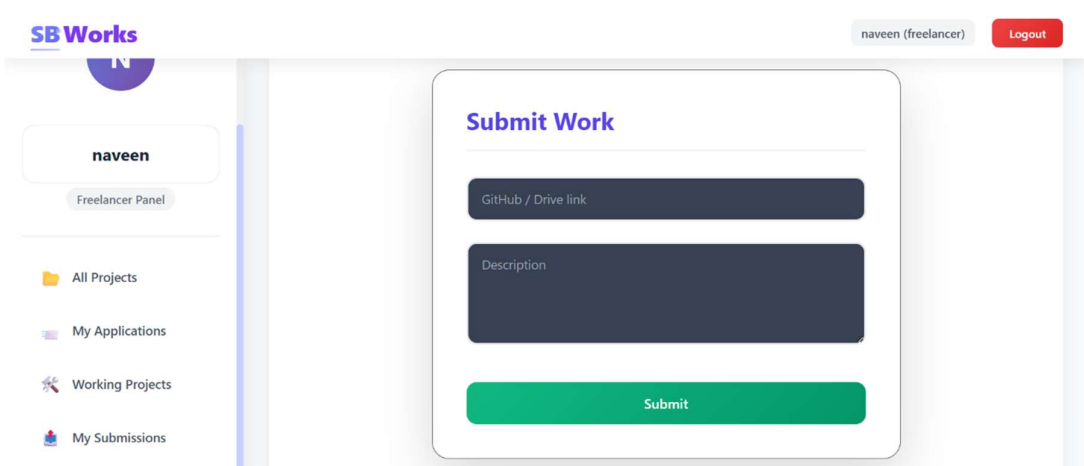


Fig 11: Freelancer submitting the work

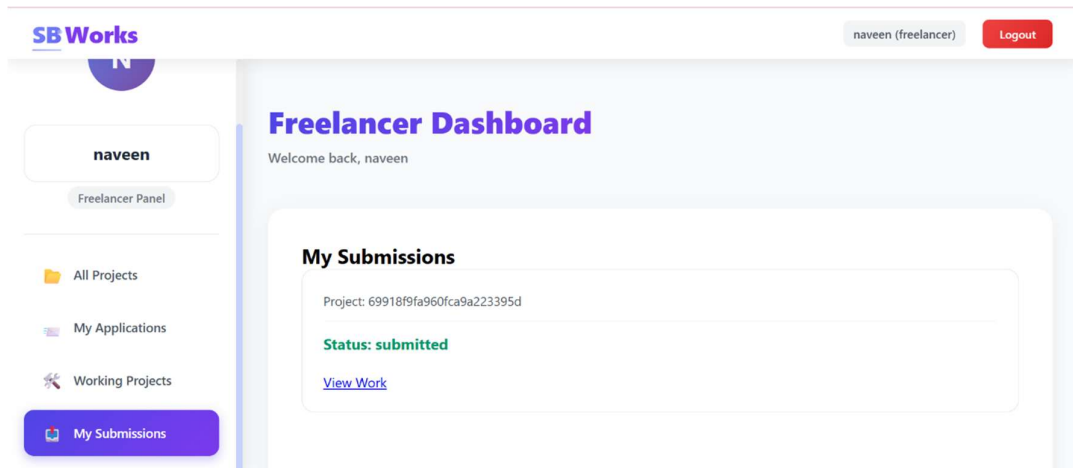


Fig 12: Freelancer submitting the project

10. Testing

Overview of Testing Strategy

Testing in this project was conducted to ensure reliability, security, performance, and functional correctness of the system. A structured testing approach was followed, covering:

- Unit Testing
- Integration Testing
- API Testing
- UI Testing
- Manual Testing
- Security Testing

The objective was to validate that each component works independently and that all modules function correctly when integrated.

10.1 Unit Testing

Purpose

Unit testing was performed to verify that individual functions and modules behave as expected in isolation.

Backend Unit Testing

Backend logic such as:

- Authentication controllers
- Authorization middleware
- Database operations
- Validation functions were tested independently.

Tools Used

- **Jest** – JavaScript testing framework
- **Supertest** – HTTP assertions for API endpoints

Unit tests ensured:

- Correct password hashing
- Proper JWT token generation
- Validation errors returned correctly
- Database queries return expected results

10.2 Integration Testing

Purpose

Integration testing verified that different modules of the system work together properly.

Scope

- Login flow (request → authentication → token generation → response)
- Protected routes with middleware
- Role-based authorization checks
- CRUD operations for projects and users

This ensured that interactions between controllers, middleware, and the database were functioning as expected.

10.3 API Testing

Purpose

API testing validated backend endpoints independently of the frontend.

Tool Used

- **Postman**

Using Postman, the following were tested:

- User registration endpoint
- Login endpoint
- Project creation endpoint
- Update and delete endpoints
- Admin-only endpoints

Tests verified:

- Correct HTTP status codes (200, 201, 400, 401, 403, 500)
- Proper JSON responses

- Token validation
- Error handling

API testing ensured backend reliability before frontend integration.

10.4 Frontend Testing

Manual UI Testing

Manual testing was conducted to verify:

- Form validation
- Button functionality
- Navigation between pages
- Conditional rendering based on roles
- Dashboard data display

Each UI component was tested under different user roles (Admin and Freelancer).

Functional Testing

The following user scenarios were tested:

1. User Registration
2. User Login
3. Create Project
4. Apply for Project
5. Edit Project
6. Delete Project
7. Admin Managing Users

Each scenario was tested from start to finish to ensure complete workflow correctness.

10.5 Authentication & Authorization Testing

Special attention was given to security-related testing.

Test Cases Included:

- Accessing protected routes without token
- Accessing protected routes with invalid token
- Accessing admin routes as normal user
- Token expiration behavior
- Logout functionality

Expected results:

- 401 Unauthorized for invalid or missing tokens

- 403 Forbidden for insufficient permissions

This ensured robust access control enforcement.

10.6 Database Testing

Database testing verified:

- Data is stored correctly
- Passwords are hashed
- Role values are stored accurately
- Cascade delete or update operations behave correctly

Test data was inserted and verified directly from the database to ensure integrity.

10.7 Performance Testing

Basic performance testing was conducted to evaluate:

- API response time
- Dashboard loading speed
- Concurrent user handling

Results showed acceptable response times under normal load conditions.

11. Screenshots or Demo

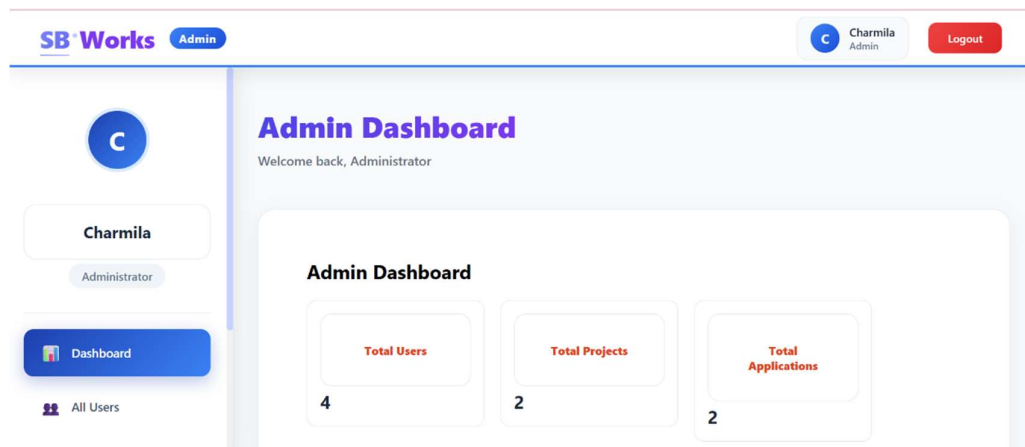


Fig 13: Admin dashboard

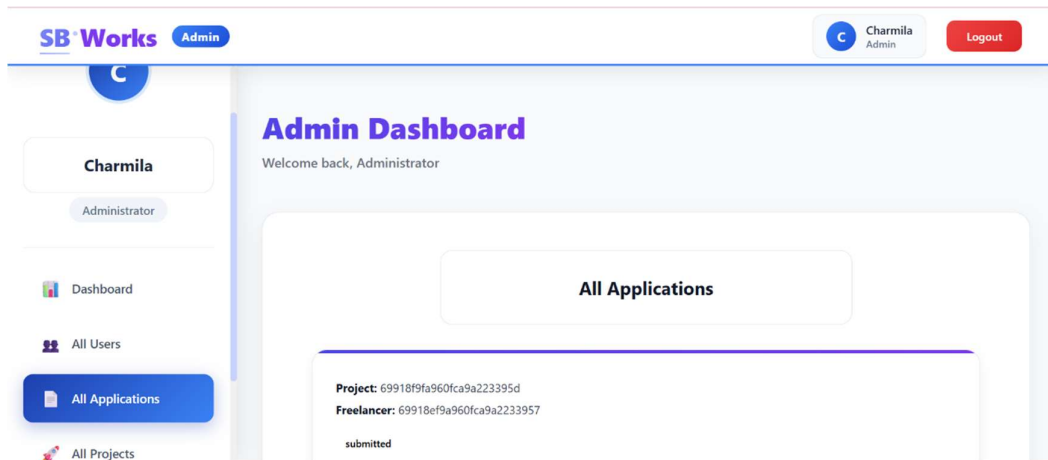


Fig 14: Admin viewing the applications

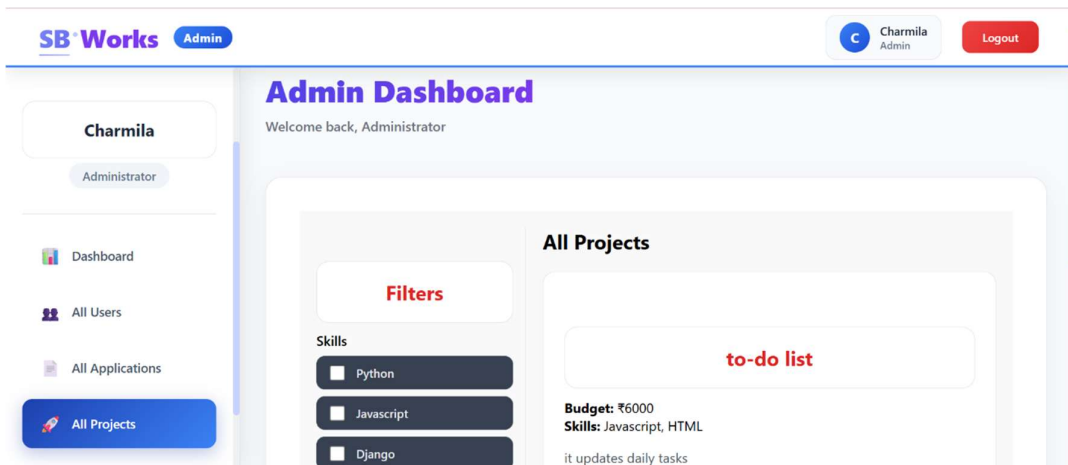


Fig 15: All the projects viewed by Admin

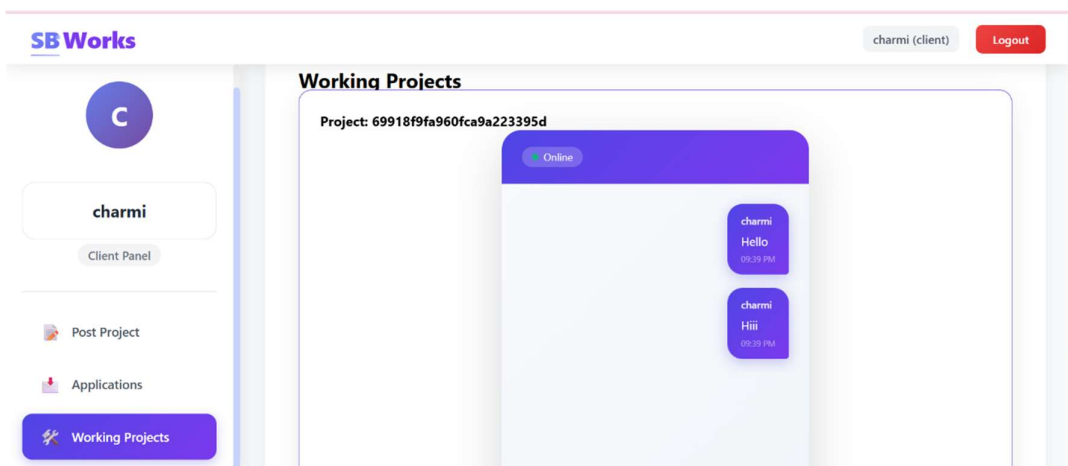


Fig 16: Chat Conversation within the website

12. Known Issues

- **Token Expiry Without Auto Refresh**

The system does not implement automatic refresh tokens, so users must log in again after token expiration.

This may interrupt active sessions unexpectedly.

- **Limited Password Recovery Options**

There is no built-in forgot password or reset password functionality. Users cannot recover accounts without administrative assistance.

- **No Real-Time Updates**

Dashboard updates require manual refresh to reflect new data. The absence of live notifications affects real-time interaction.

- **Basic Role Management**

Only predefined roles such as Admin and Freelancer are supported. Granular permission control is not currently implemented.

- **UI Responsiveness Limitations**

Some data tables may require horizontal scrolling on smaller screens. Mobile optimization can be further improved for better usability.

13. Future Enhancements

- **Refresh Token Mechanism**

Implementing refresh tokens will allow seamless session renewal. This will improve user experience and enhance security.

- **Email Verification and Password Reset**

Adding email verification and secure reset links will strengthen account protection. This will improve reliability and user trust.

- **Real-Time Notifications**

Integrating WebSocket-based notifications will enable instant updates. Users will receive alerts for new applications and status changes.

- **Advanced Role-Based Access Control**

Introducing customizable roles and permission settings will improve flexibility. This will make the system suitable for enterprise-level usage.

- **Payment Gateway Integration**

Adding secure payment processing will support milestone-based transactions. This will transform the platform into a complete freelance management system.