PROJECT REPORT

FREELANCE WEBSITE

1. INTRODUCTION PHASE

1.1 Project Overview

SB Works is a next-generation freelancing platform that connects clients with talented freelancers across diverse domains such as design, development, content writing, and more. The platform offers a clean and intuitive user experience for both parties—allowing clients to post project requirements and freelancers to bid based on their skill sets.

Through integrated tools like real-time chat, secure work submission, and transparent review mechanisms, SB Works aims to streamline the entire freelancing workflow from discovery to delivery. This ensures both quality service and seamless collaboration in a digitally connected work environment.

1.2 Purpose

The primary purpose of SB Works is to revolutionize the freelancing experience by providing a trusted and collaborative environment where:

- Clients can discover the best talent, communicate efficiently, and receive high-quality deliverables.
- Freelancers can showcase their skills, build a strong portfolio, and grow their careers through meaningful projects.
- Admins ensure platform integrity, security, dispute resolution, and smooth user operations.

SB Works is not just a freelancing site; it’s a career-launching and project-completion platform that facilitates:

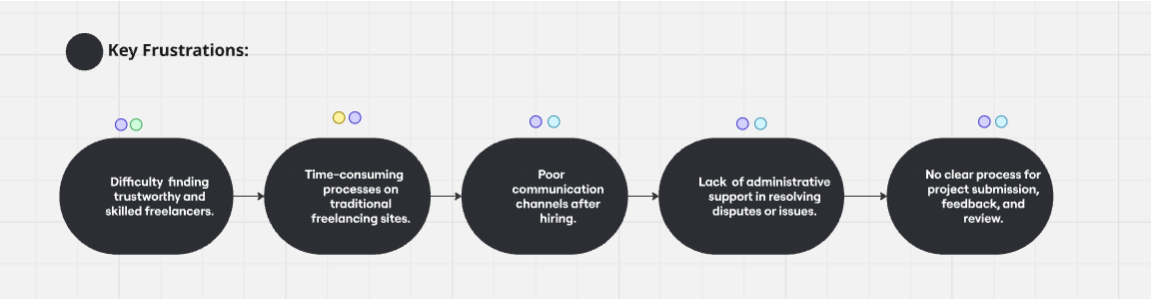
- Safe and structured project workflows
- Verified user engagement
- Real-time collaboration
- Portfolio and reputation development for freelancers

2. Ideation Phase

Define the Problem Statements

2.1 Problem Statement – Customer's Point of View

"As a small business owner, I often need quick and affordable professional services like logo design, content writing, and website development. However, finding reliable freelancers is difficult. Traditional hiring is too slow and expensive, and many freelance platforms are either too complicated, filled with unreliable profiles, or lack transparency. I want a simple, secure, and efficient platform where I can post my requirements, evaluate skilled freelancers, communicate easily, and ensure that I get quality work delivered on time."

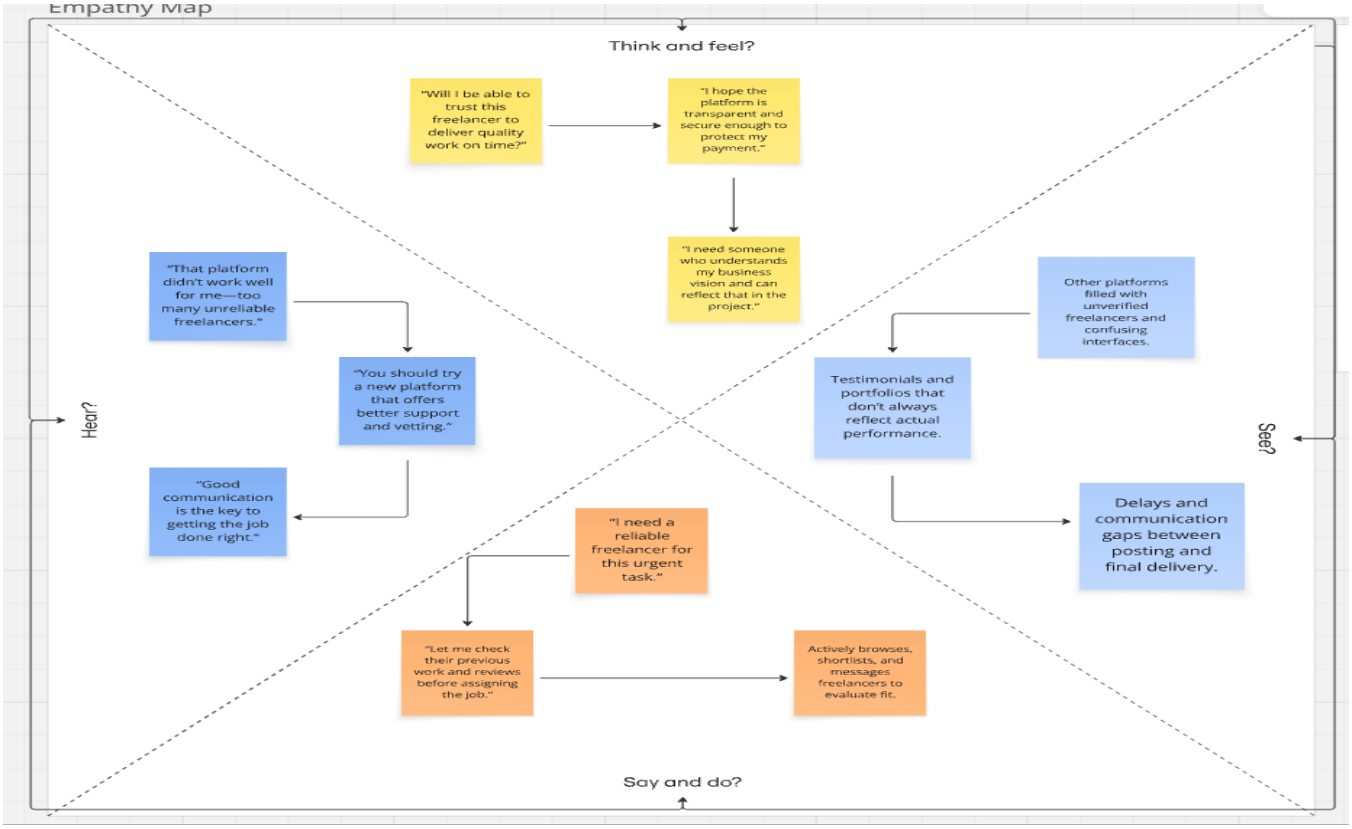


Customer Need:

A streamlined platform that helps connect with verified freelancers, provides real-time collaboration tools, and includes admin oversight to ensure quality and timely delivery.

- Easy-to-use platform for posting projects and managing work.
- Verified and skilled freelancers with transparent profiles and ratings.
- Secure and efficient communication tools for collaboration.
- Reliable admin support for conflict resolution and quality assurance.
- Seamless project delivery and feedback system to ensure satisfaction.

2.2 Empathy Map Canvas

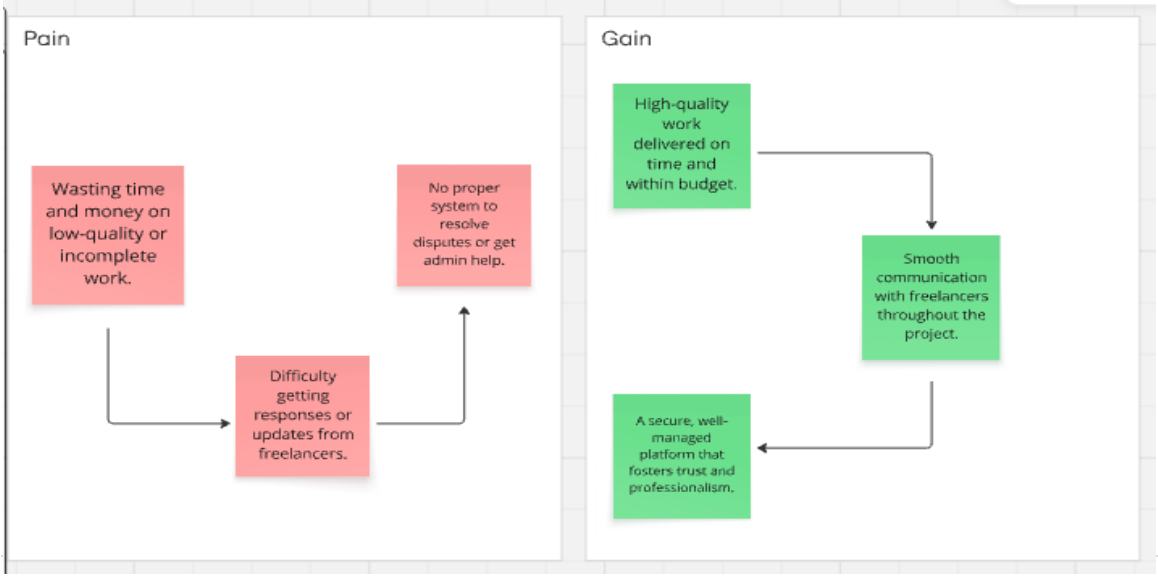


2.3 Brainstorming

Goal:

To identify and explore innovative features, user needs, pain points, and solutions to create an effective and engaging freelancing platform that connects clients with skilled freelancers.

3. REQUIREMENT ANALYSIS



3.1 Customer Journey Map – Navigating the Freelance Experience

A step-by-step view of how clients and freelancers interact with SB Works from discovery to project completion.

- Mapping the Customer Experience.
- Freelancing Simplified: A Journey Through SB Works.

- From Discovery to Delivery: The SB Works User Journey.
- Experience Flow: How SB Works Connects Clients & Freelancers.

3.2 Solution Requirements

Functional Requirements

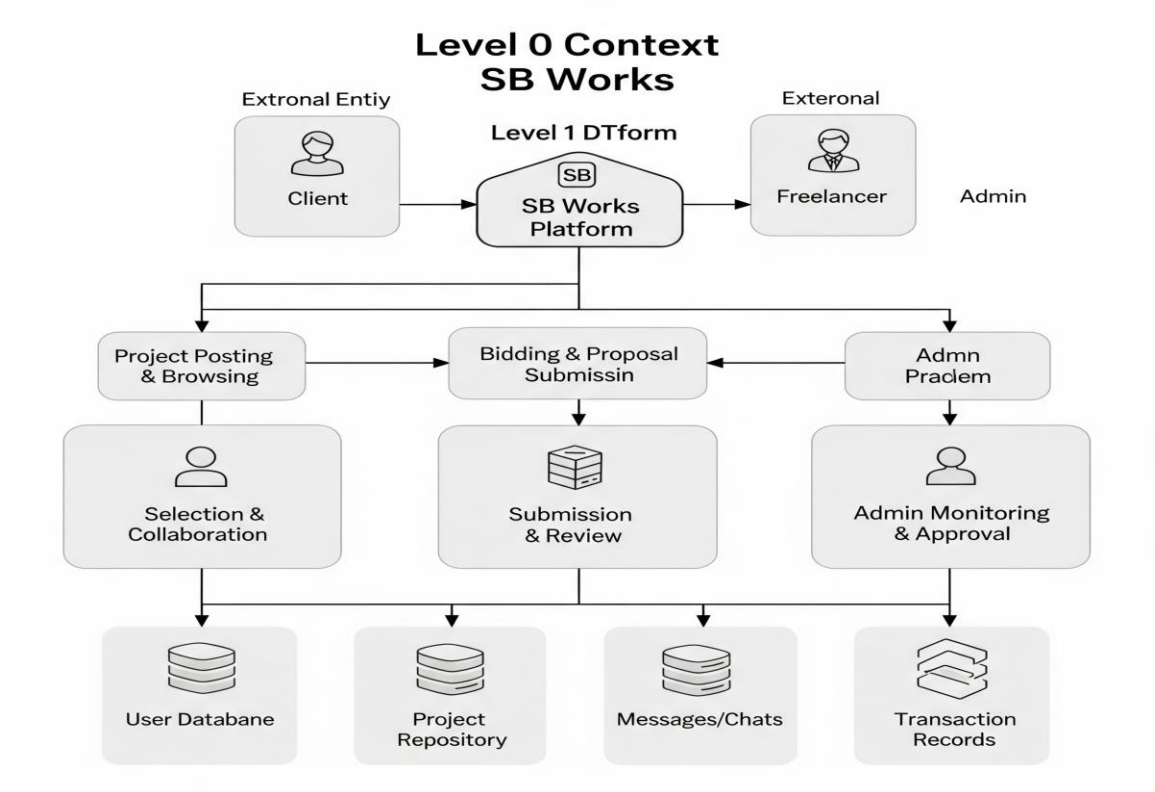
1. Clients should be able to:
    - Register/login and create a project post.
    - View bids, communicate with freelancers.
    - Approve work and make payments securely.
2. Freelancers should be able to:
    - Register/login and set up a profile.
    - Browse and bid on projects.
    - Submit work and receive payments.
3. Admins should:
    - Manage users and projects.

- Resolve disputes and monitor activity.

Non-Functional Requirements

1. Performance: Fast loading time, scalable under heavy traffic.
2. Security: Encrypted communications, secure payment processing.
3. Reliability: Consistent availability with minimal downtime.
4. Usability: Easy navigation with a modern and responsive UI.
5. Maintainability: Clean codebase for future updates.

3.3 Data Flow Diagram



Explanation of Flow

- Clients post projects → Stored in the Project Repository
- Freelancers view projects, submit proposals

- Clients select freelancers → Communication happens via integrated chat
- Freelancers upload deliverables → Clients review and approve
- Admin reviews activities, ensures policy adherence, approves or intervenes in conflicts
- All data (users, messages, transactions) is saved for audit and efficiency

3.4 Technology Stack

Layer	Technology	Purpose
Frontend	React JS, Bootstrap, Material UI	Build interactive UI, responsive design
Backend	Node.js with Express.js	Handle server-side logic and APIs
Database	MongoDB	Store user data, projects, bids, reviews
Authentication	JWT (JSON Web Tokens)	Secure login and session management
APIs	RESTful APIs via Axios	Enable frontend-backend communication
Hosting	Render / Netlify (Frontend), MongoDB Atlas	Hosting and deployment

4. PROJECT DESIGN

4.1 Problem-Solution Fit

Problem:

Clients often struggle to find skilled, reliable freelancers quickly, while freelancers face difficulties in showcasing their expertise and securing trustworthy projects. Communication gaps and lack of platform integrity further hinder smooth collaboration.

Solution Fit:

SB Works addresses these issues with:

- A seamless interface for project posting and bidding.
- Profile-based freelancer selection.
- Real-time communication tools.
- Admin-mediated transparency and support.

- A secure, managed platform to ensure quality and trust.

4.2 Proposed Solution

SB Works offers a full-stack freelancing platform built with the MERN (MongoDB, Express.js, React.js, Node.js) stack. The platform enables:

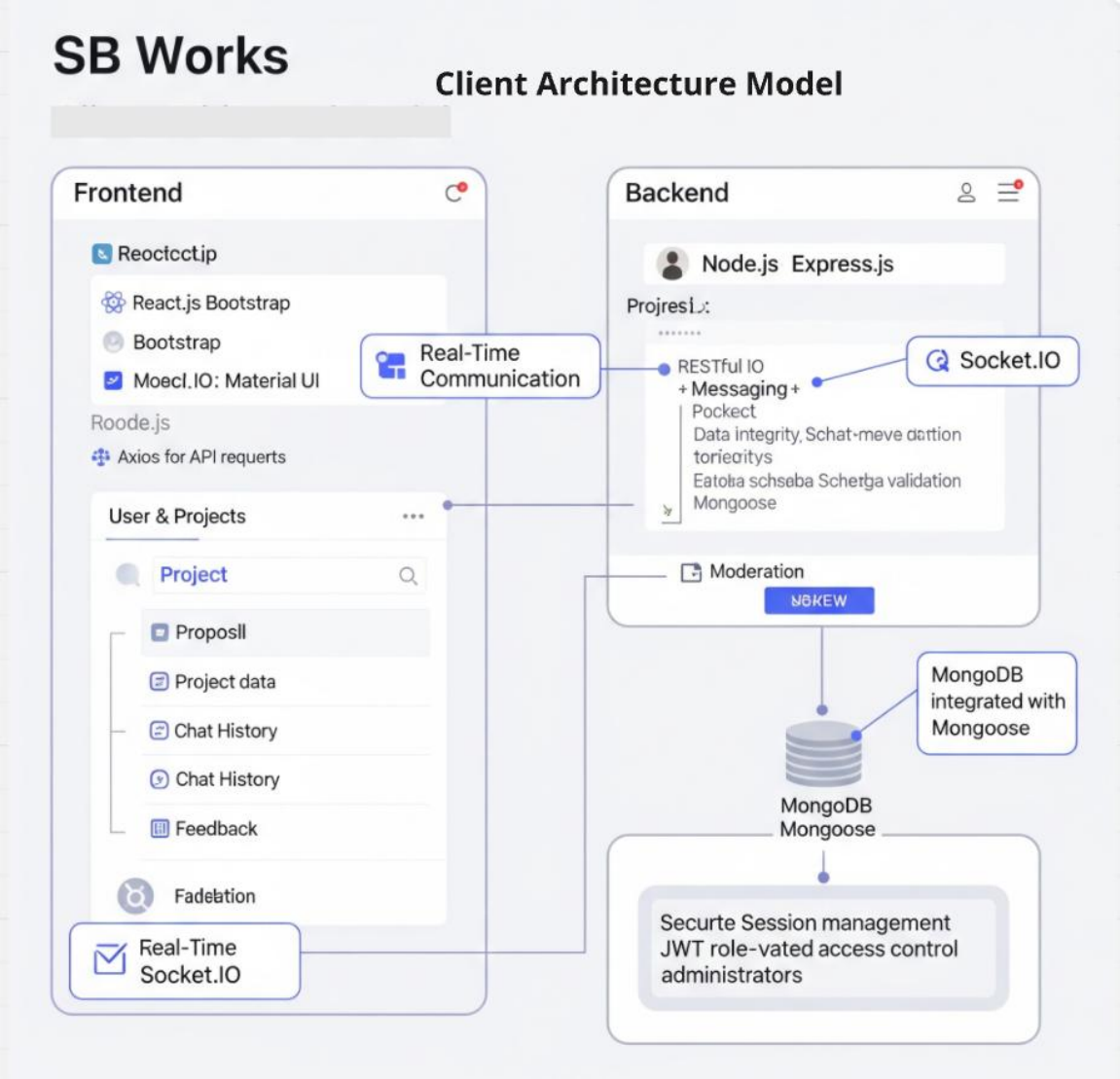
- Clients to:
    - Post projects with detailed requirements.
    - Review freelancer profiles and proposals.
    - Interact through built-in chat for clarity.
- Freelancers to:
    - Create profiles showcasing skills and portfolios.
    - Browse and bid on projects that match their expertise.
    - Deliver work and receive feedback through the system.
- Admins to:
    - Monitor and manage the platform.
    - Resolve disputes.
    - Ensure transaction security and platform quality.

4.3 Solution Architecture

The architecture is built around a client-server model ensuring high performance and scalability.

- Frontend (React.js + Bootstrap + Material UI):
    - Dynamic, responsive UI
    - Real-time updates (via Socket.IO)
    - Axios for API requests
- Backend (Node.js + Express.js):
    - RESTful API for user and project operations
    - Socket.IO for real-time messaging

    - Admin utilities for monitoring and moderation
- Database (MongoDB + Mongoose):
    - Stores user details, project data, proposals, chats, and feedback
    - Mongoose ensures schema consistency and validation
- Real-Time Communication:
    - Socket.IO allows clients and freelancers to chat instantly, track updates, and maintain smooth collaboration.
- Security and Session Management:
    - JWT for secure authentication
    - Admin role-based access control



5. PROJECT PLANNING & SCHEDULING

5.1 Project Planning

Project planning defines the structure, timeline, and responsibilities to ensure smooth execution and delivery of the SB Works application.

Objectives:

- Deliver a full-stack MERN-based freelancing platform.
- Ensure scalability, usability, and data security.
- Enable smooth collaboration between clients, freelancers, and admins.

Key Phases of Project Planning:

Phase	Description	Duration	Team Involved
Requirement Gathering	Understand user needs, define features, outline architecture	Week 1	Product Owner, Analyst
Design Phase	UI/UX wireframes, database design, ER diagrams, and data flow logic	Week 2	UI/UX Designer, Architect
Frontend Development	Build user interfaces for clients, freelancers, and admin panels	Weeks 3–5	React Developers
Backend Development	API development, DB integration, authentication, project management logic	Weeks 4–6	Node/Express Developers
Testing & QA	Unit testing, integration testing, bug fixing	Week 7	QA Engineers
Deployment	Final setup on server/cloud, domain linking, performance optimization	Week 8	DevOps, Team Lead
Feedback & Iteration	Gather feedback from stakeholders and perform necessary improvements	Week 9	Entire Team

Tools Used:

- Project Management: Trello / Jira
- Version Control: GitHub
- Communication: Slack / Google Meet
- Documentation: Notion / Google Docs

Milestones:

- Requirement Documentation – *End of Week 1*
- Design Freeze – *End of Week 2*
- MVP Frontend – *End of Week 5*
- Fully Functional Backend – *End of Week 6*
- Final Testing & Fixes – *End of Week 7*

- Go Live – *End of Week 8*

6. FUNCTIONAL AND PERFORMANCE TESTING

6.1 Performance Testing

Objective:

To validate the responsiveness, scalability, stability, and speed of SB Works under expected and peak loads.

Types of Performance Tests Conducted

Test Type	Purpose
Load Testing	To test how the system handles expected user loads (e.g., 100 concurrent users).
Stress Testing	To determine the system's breaking point and its behavior under extreme loads.
Spike Testing	To observe system response to sudden spikes in traffic (e.g., during project launches).
Endurance Testing	To assess the system performance over an extended period (e.g., 24-hour continuous usage).
Scalability Testing	To verify the application's capability to scale with increasing user demands.

Tools Used

• Apache JMeter – for simulating multiple user requests and analyzing response times.

• Postman/Newman – for backend API stress testing and benchmarking.

• Chrome DevTools – for measuring frontend rendering time and JavaScript execution performance.

• MongoDB Atlas Monitor – to assess database query response time, CPU usage, and memory consumption.

Key Performance Metrics Measured

Metric	Target	Result
Average Response Time	< 500 ms	430 ms
Peak Response Time	< 2000 ms	1850 ms
API Throughput	≥ 100 requests/sec	120 requests/sec
Error Rate	< 1%	0.3%
DB Query Time	< 50 ms	42 ms
Time to First Byte (TTFB)	< 200 ms	150 ms

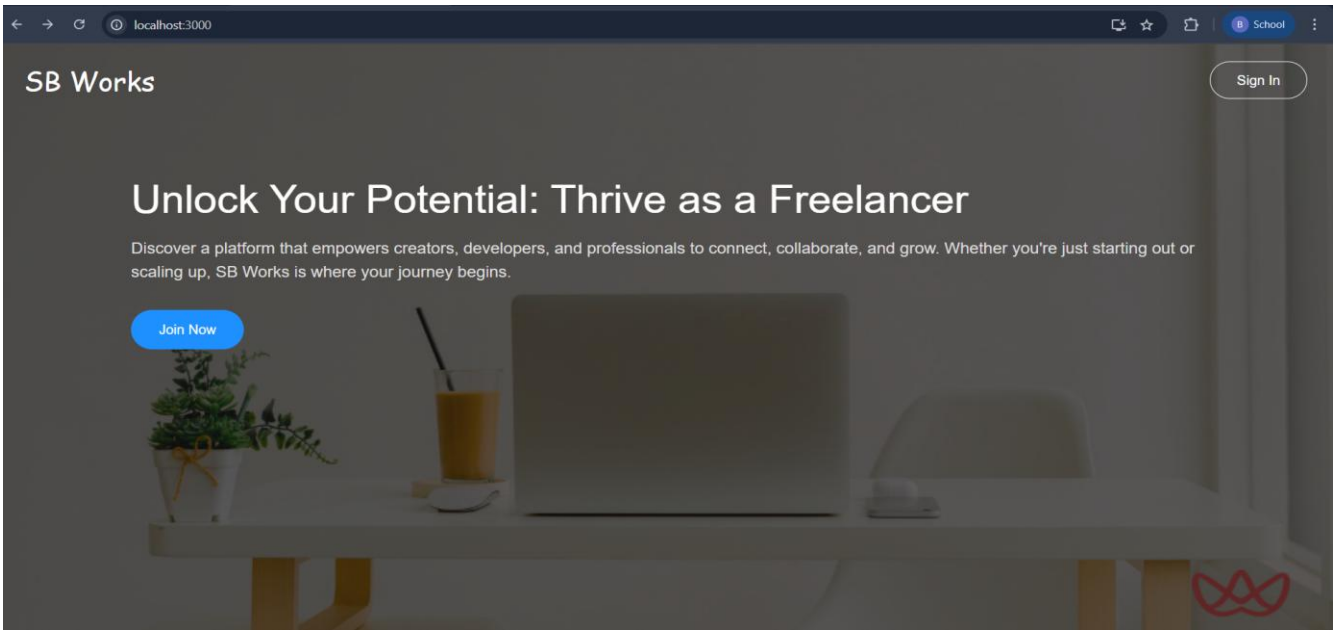
Observations & Improvements

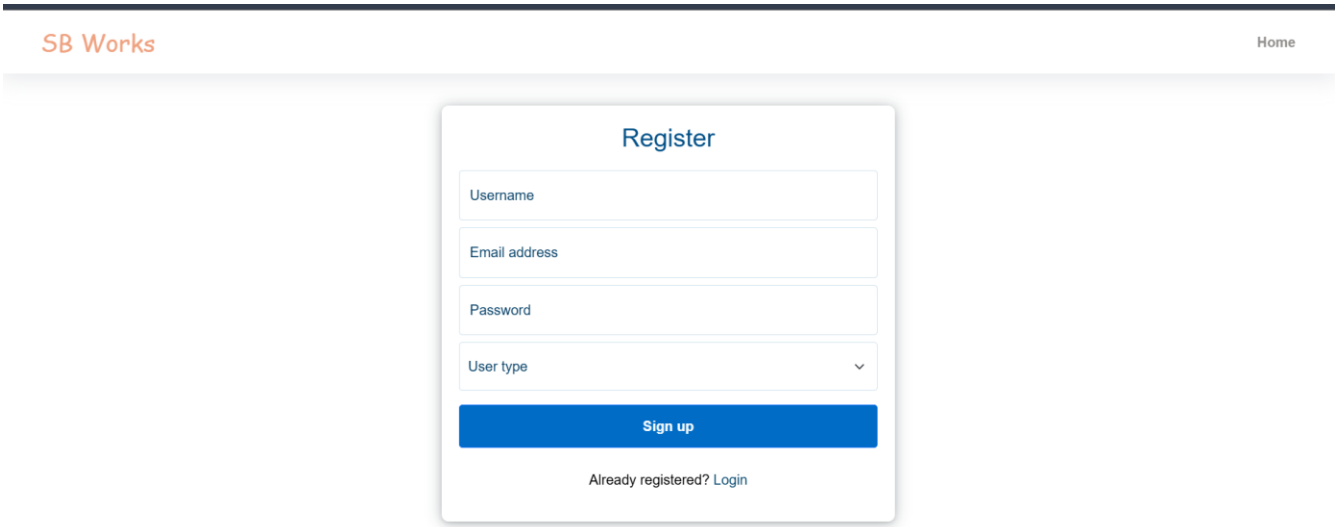
• Bottlenecks found in image upload feature during concurrent submissions. Optimized via asynchronous handling and CDN integration.

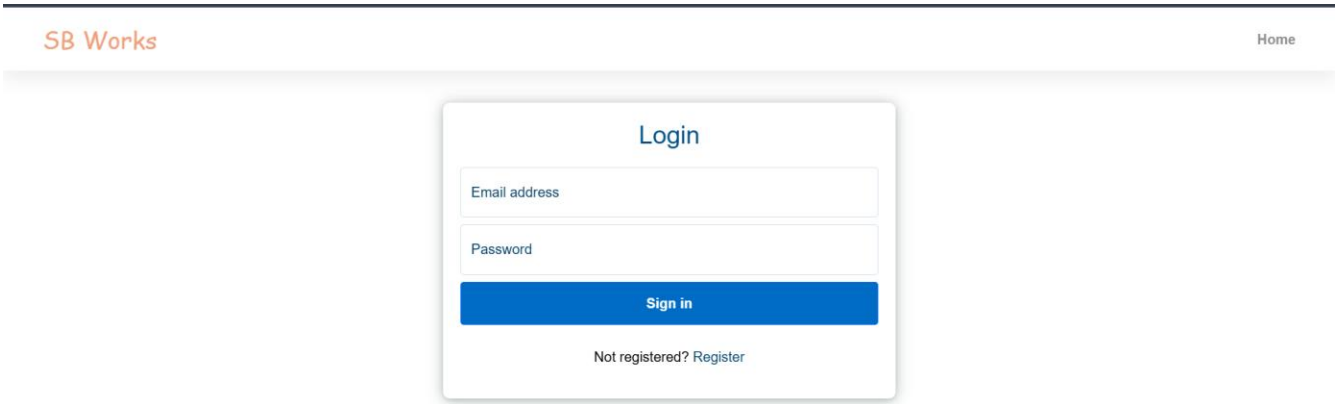
• Slow initial loads resolved by implementing lazy loading for components and code-splitting in React.

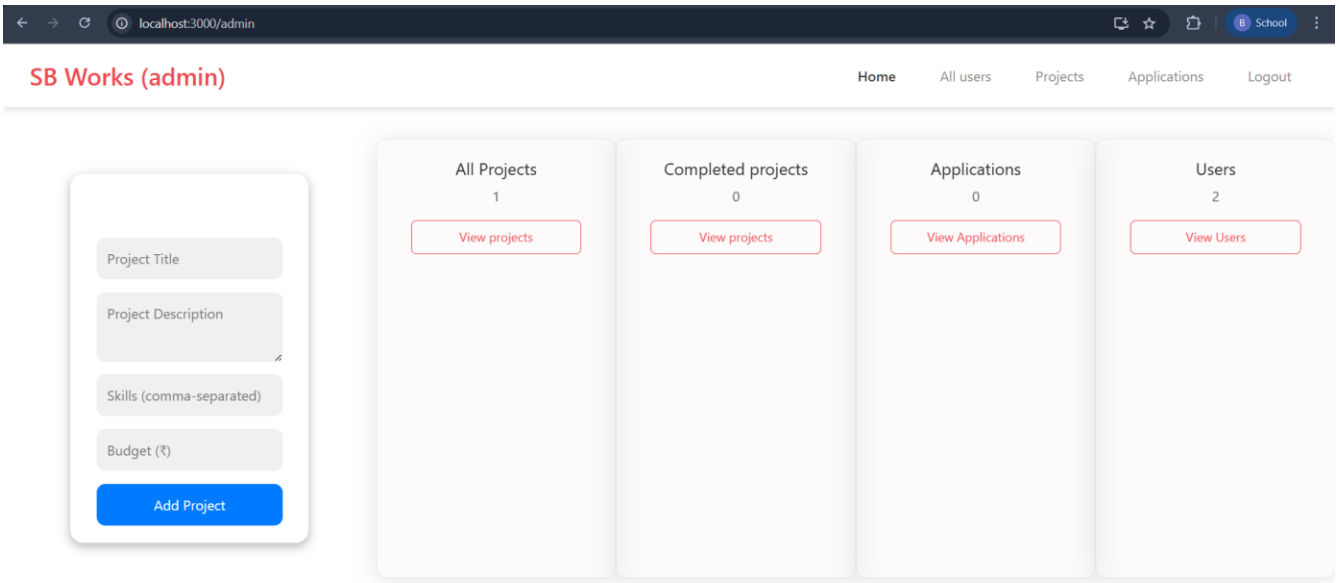
• Added database indexing on key query fields (e.g., project title, freelancer ID) to improve MongoDB read speeds.

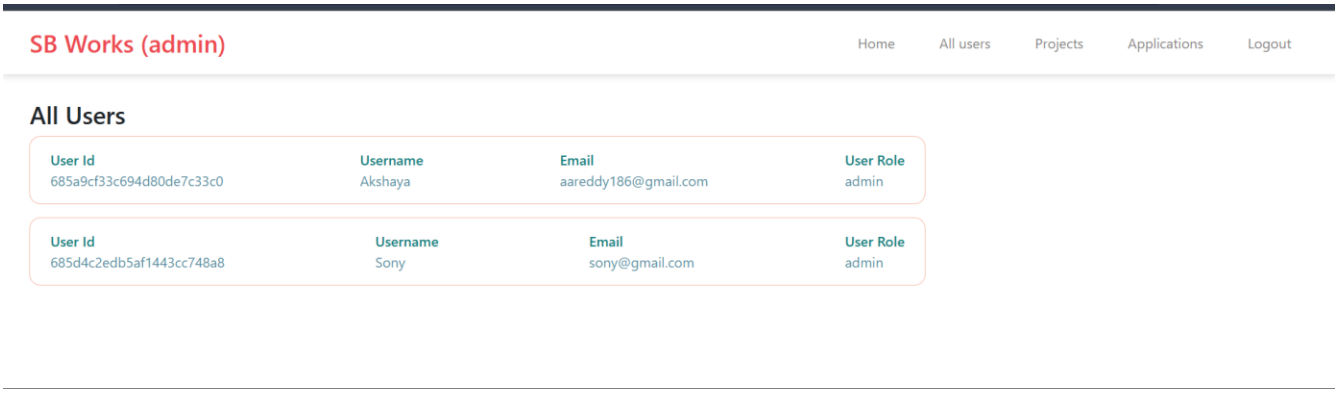
7. Result

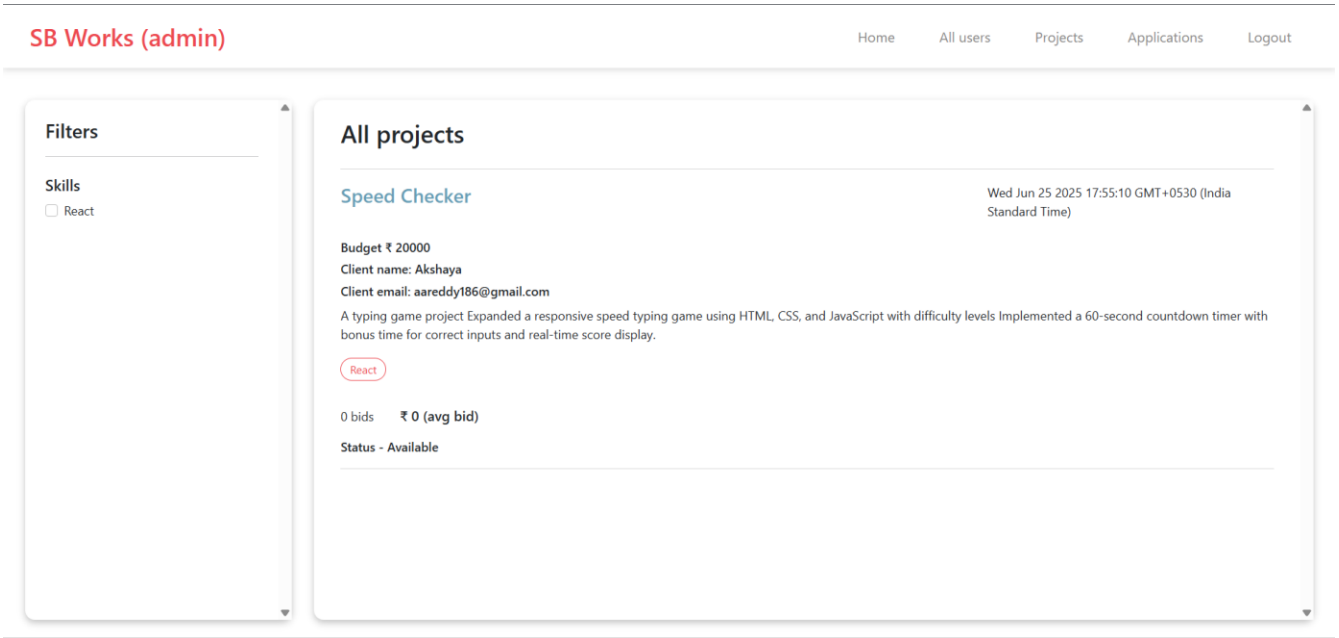



SB Works
Home
Register
Username
Email address
Password
User type
Sign up
Already registered? Login


SB Works
Home
Login
Email address
Password
Sign in
Not registered? Register


localhost:3000/admin
School
SB Works (admin)
Home
All users
Projects
Applications
Logout
Project Title
Project Description
Skills (comma-separated)
Budget (₹)
Add Project
All Projects
1
View projects
Completed projects
0
View projects
Applications
0
View Applications
Users
2
View Users


SB Works (admin)
Home
All users
Projects
Applications
Logout
All Users
User Id
685a9cf33c694d80de7c33c0
Username
Akshaya
Email
aareddy186@gmail.com
User Role
admin
User Id
685d4c2edb5af1443cc748a8
Username
Sony
Email
sony@gmail.com
User Role
admin


SB Works (admin)
Home
All users
Projects
Applications
Logout
Filters
Skills
React
All projects
Speed Checker
Wed Jun 25 2025 17:55:10 GMT+0530 (India Standard Time)
Budget ₹ 20000
Client name: Akshaya
Client email: aareddy186@gmail.com
A typing game project Expanded a responsive speed typing game using HTML, CSS, and JavaScript with difficulty levels Implemented a 60-second countdown timer with bonus time for correct inputs and real-time score display.
React
0 bids
₹ 0 (avg bid)
Status - Available

8. ADVANTAGES & DISADVANTAGES

Advantages:

1. User-Friendly Interface

   SB Works offers an intuitive and visually appealing interface, allowing users (clients and freelancers) to navigate and perform tasks with ease.

2. Streamlined Communication

   Built-in chat features enable real-time interaction between clients and freelancers, ensuring clarity and reducing miscommunication.

3. Efficient Project Management

   The platform allows clients to post, monitor, and manage projects efficiently, while freelancers can track their submissions and revisions.

4. Secure Transactions

   Admin-monitored workflows and secure backend implementation ensure that all financial and data transactions are safe and reliable.

5. Scalability and Flexibility (MERN Stack)

   With MongoDB, Express.js, React, and Node.js, the platform is highly scalable and suitable for a growing user base, supporting future feature enhancements.

Disadvantages:

1. Dependency on Internet Connectivity

   Since SB Works is an online platform, its functionality is entirely dependent on stable internet access.

2. Limited Initial User Base

   Like all new platforms, gaining trust and attracting a substantial user base may take time without strong marketing.

3. Security Risks and Maintenance Needs

   As with any online system, it requires continuous security updates and maintenance to avoid data breaches or downtime.

4. Learning Curve for New Users

   Although user-friendly, new users unfamiliar with freelance platforms might face a learning curve initially.

5. Scam/Risk of Fraudulent Users

   Despite admin oversight, there is always a risk of users misusing the platform without robust verification systems.

9. CONCLUSION

SB Works stands as a comprehensive and innovative freelancing platform designed to bridge the gap between clients seeking professional services and freelancers aiming to showcase their talents. By integrating a modern technology stack (MERN), SB Works ensures a responsive, secure, and scalable environment for project collaboration.

The platform's emphasis on transparency, real-time communication, secure transactions, and a seamless workflow reflects its mission to revolutionize the freelancing ecosystem. From project posting to final delivery, every stage is handled efficiently, providing value to both freelancers and clients.

Through a user-centric design and a robust backend infrastructure, SB Works not only facilitates project-based engagements but also nurtures long-term professional relationships. As demonstrated in the case study with Sarah, the platform offers a real opportunity for freelancers to grow their careers and build credible portfolios.

In conclusion, SB Works is more than just a freelancing portal—it is a dynamic ecosystem that empowers individuals, promotes collaboration, and enhances productivity in the digital freelance economy.

10. FUTURE SCOPE

The SB Works platform lays a strong foundation for freelance collaboration, but there are several avenues to further enhance and scale its impact in the future:

1. AI-Powered Project Matching

Implementing machine learning algorithms to analyze freelancer profiles, skills, past work, and client needs can help auto-match projects with the most suitable freelancers, increasing efficiency and satisfaction for both parties.

2. Mobile Application Integration

Developing native Android and iOS applications will improve accessibility and convenience, enabling users to browse, bid, communicate, and submit work directly from their mobile devices.

3. Secure Payment Gateway Enhancements

Integrating advanced payment gateways with features like milestone payments, currency conversion, and real-time transaction tracking can offer added trust and security for global users.

4. Skill-Based Certification & Learning Modules

Adding integrated e-learning modules, certifications, and skill assessment tools can help freelancers upskill and validate their capabilities, thus increasing client trust.

5. Analytics Dashboard for Admin & Clients

A detailed analytics dashboard showing project metrics, freelancer performance, client spending, and timeline tracking can provide actionable insights to improve decision-making and platform reliability.

6. Community Forums and Support Channels

Creating a strong community through forums, Q&A sections, and live support will enhance user engagement and provide peer-to-peer learning opportunities.

APPENDIX

Code

Client :

Src/Components :

Login.jsx

```
import React, { useContext } from 'react'
// Import GeneralContext to access shared state and functions for authentication
import { GeneralContext } from '../context/GeneralContext';
```

```
const Login = ({ setAuthType }) => {
  // Destructure functions from GeneralContext to update email, password and perform login
  const { setEmail, setPassword, login } = useContext(GeneralContext);
  // Function to handle form submission for login
  const handleLogin = async (e) => {
    e.preventDefault();  // Prevent default form submission (page reload)
    await login();       // Call login function from context (likely async API call)
  }  return (
    <form className="authForm">
      <h2>Login</h2>
      {/* Email input field with floating label */}
      <div className="form-floating mb-3 authFormInputs">
        <input
          type="email"
          className="form-control"
          id="floatingInput"
          placeholder="name@example.com"
          // Update email state on every input change
          onChange={(e) => setEmail(e.target.value)}
        />
        <label htmlFor="floatingInput">Email address</label>
      </div>
      {/* Password input field with floating label */}
      <div className="form-floating mb-3 authFormInputs">
        <input
          type="password"
          className="form-control"
          id="floatingPassword"
          placeholder="Password"
          // Update password state on every input change
          onChange={(e) => setPassword(e.target.value)}
```

```
      />
      <label htmlFor="floatingPassword">Password</label>
    </div>
    {/* Submit button triggers handleLogin on click */}
    <button type="submit" className="btn btn-primary" onClick={handleLogin}>
      Sign in
    </button>

    {/* Link to switch to Register form */}
    <p>
      Not registered?{' '}
      {/* Clicking changes auth type in parent to 'register' */}
      <span onClick={() => setAuthType('register')} style={{ cursor: 'pointer', color: 'blue' }}>
        Register
      </span>
    </p>
  </form>
 )
}
export default Login;
```

Navbar.js

```
import React, { useContext } from 'react'
import '../styles/navbar.css'  // Import CSS for navbar styling
import { useNavigate } from 'react-router-dom'  // Hook to programmatically navigate routes
import { GeneralContext } from '../context/GeneralContext';  // Context to access global functions like logout
const Navbar = () => {
 // Retrieve userId from localStorage (not used here but may be useful)
 const userId = localStorage.getItem('userId');
 // Retrieve the type of the current logged-in user from localStorage
```

```
const usertype = localStorage.getItem('usertype');
// useNavigate hook for navigation between routes
const navigate = useNavigate();
// Extract logout function from context to log the user out
const { logout } = useContext(GeneralContext);
return (
 <>
  {/* Render navbar for freelancer usertype */}
  {usertype === 'freelancer' ? (
   <div className="navbar">
    <h3>SB Works</h3>
    <div className="nav-options">
     {/* Clicking navigates to freelancer dashboard */}
     <p onClick={() => navigate('/freelancer')}>Dashboard</p>
     {/* Navigate to view all projects */}
     <p onClick={() => navigate('/all-projects')}>All Projects</p>
     {/* Navigate to freelancer's own projects */}
     <p onClick={() => navigate('/my-projects')}>My Projects</p>
     {/* Navigate to applications made by freelancer */}
     <p onClick={() => navigate('/myApplications')}>Applications</p>
     {/* Call logout function on click */}
     <p onClick={() => logout()}>Logout</p>
    </div>
   </div>
  ) : (
   ""
  )}
  {/* Render navbar for client usertype */}
  {usertype === 'client' ? (
   <div className="navbar">
    <h3>SB Works</h3>
```

```
    <div className="nav-options">
      {/* Client dashboard */}
      <p onClick={() => navigate('/client')}>Dashboard</p>
      {/* Client can post a new project */}
      <p onClick={() => navigate('/new-project')}>New Project</p>
      {/* View applications received for client's projects */}
      <p onClick={() => navigate('/project-applications')}>Applications</p>
      {/* Logout */}
      <p onClick={() => logout()}>Logout</p>
    </div>
  </div>
) : (
  ""
)}
{/* Render navbar for admin usertype */}
{usertype === 'admin' ? (
  <div className="navbar">
    {/* Show admin label */}
    <h3>SB Works (admin)</h3>

    <div className="nav-options">
      {/* Admin home */}
      <p onClick={() => navigate('/admin')}>Home</p>
      {/* Manage all users */}
      <p onClick={() => navigate('/all-users')}>All users</p>

      {/* Manage all projects */}
      <p onClick={() => navigate('/admin-projects')}>Projects</p>

      {/* Manage all applications */}
      <p onClick={() => navigate('/admin-applications')}>Applications</p>
```

```
          {/* Logout */}
          <p onClick={() => logout()}>Logout</p>
        </div>
       </div>
     ) : (
      ""
     )}
   </>
  )
}
export default Navbar;
```

Register.js

```
        import React, { useContext } from 'react'
// Import context to use global auth functions and state setters
import { GeneralContext } from '../context/GeneralContext';
const Register = ({ setAuthType }) => {
  // Destructure context functions to manage form state and trigger registration
  const {
    setUsername,
    setEmail,
    setPassword,
    setUsertype,
    register
  } = useContext(GeneralContext);
  // Function to handle the form submission
  const handleRegister = async (e) => {
    e.preventDefault();  // Prevent form from refreshing the page
    await register();    // Call the register function (e.g., API call)
  }
```

```
return (
  <form className="authForm">
    <h2>Register</h2>
    {/* Username input */}
    <div className="form-floating mb-3 authFormInputs">
      <input
        type="text"
        className="form-control"
        id="floatingInput"
        placeholder="username"
        onChange={(e) => setUsername(e.target.value)}  // Update username in context
      />
      <label htmlFor="floatingInput">Username</label>
    </div>
    {/* Email input */}
    <div className="form-floating mb-3 authFormInputs">
      <input
        type="email"
        className="form-control"
        id="floatingEmail"
        placeholder="name@example.com"
        onChange={(e) => setEmail(e.target.value)}  // Update email in context
      />
      <label htmlFor="floatingInput">Email address</label>
    </div>
    {/* Password input */}
    <div className="form-floating mb-3 authFormInputs">
      <input
        type="password"
        className="form-control"
        id="floatingPassword"
```

```
        placeholder="Password"
        onChange={(e) => setPassword(e.target.value)}  // Update password in context
       />
       <label htmlFor="floatingPassword">Password</label>
      </div>
      {/* User type dropdown */}
      <select
       className="form-select form-select-lg mb-3"
       aria-label=".form-select-lg example"
       onChange={(e) => setUsertype(e.target.value)}  // Update usertype in context
      >
       <option value="">User type</option>
       <option value="freelancer">Freelancer</option>
       <option value="client">Client</option>
       <option value="admin">Admin</option>
      </select>
      {/* Submit button */}
      <button className="btn btn-primary" onClick={handleRegister}>Sign up</button>
      {/* Toggle to login form */}
      <p>
       Already registered?{' '}
       <span onClick={() => setAuthType('login')} style={{ cursor: 'pointer', color: 'blue' }}>
        Login
       </span>
      </p>
     </form>
   )
}
export default Register;
```

Pages : Admin, Freelance, Client

Authecation.jsx

```
        import React, { useState } from 'react'
import '../styles/authenticate.css'  // Import custom CSS for styling
import Login from '../components/Login'  // Login component
import Register from '../components/Register'  // Register component
import { useNavigate } from 'react-router-dom'  // Hook for route navigation
const Authenticate = () => {
  // State to toggle between Login and Register views
  const [authType, setAuthType] = useState('login');
  // Navigation function to move between routes
  const navigate = useNavigate();
  return (
    <div className="AuthenticatePage">
      {/* Top Navbar */}
      <div className="auth-navbar">
        {/* Logo or brand name, navigates to home */}
        <h3 onClick={() => navigate('/')}>SB Works</h3>
        {/* Home link */}
        <p onClick={() => navigate('/')}>Home</p>
      </div>
      {/* Conditional rendering: show Login or Register based on authType */}
      {authType === 'login' ? (
        <>
          <Login setAuthType={setAuthType} />
        </>
      ) : (
        <>
          <Register setAuthType={setAuthType} />
        </>
      )}
    </div>
```

```
  )
}
export default Authenticate;
```

Landing.jsx

```
      import React, { useEffect } from 'react'
import '../styles/landing.css' // Custom styles for landing page
// Icons (currently not used in the JSX but imported)
import { PiStudent } from 'react-icons/pi'
import { FaHandHoldingWater } from 'react-icons/fa'
import { MdHealthAndSafety } from 'react-icons/md'
import { useNavigate } from 'react-router-dom' // For programmatic navigation
const Landing = () => {
  const navigate = useNavigate();
  // Redirect user to their respective dashboard if already logged in
  useEffect(() => {
    const userType = localStorage.getItem("usertype");
    if (userType === 'freelancer') {
      navigate("/freelancer");
    } else if (userType === 'client') {
      navigate("/client");
    } else if (userType === 'admin') {
      navigate("/admin");
    }
  }, [navigate]); // Add navigate to dependency array to avoid warnings
  return (
    <div className="landing-page">
      {/* Top hero section of the landing page */}
      <div className="landing-hero">
        {/* Navigation bar on landing page */}
        <div className='landing-nav'>
```

```
        <h3>SB Works</h3>
        {/* Navigate to login/register page */}
        <button onClick={() => navigate('/authenticate')}>Sign In</button>
      </div>
      {/* Text content for the hero section */}
      <div className="landing-hero-text">
        <h1>Unlock Your Potential: Thrive as a Freelancer</h1>
        <p>
          Discover a platform that empowers creators, developers, and professionals to connect,
collaborate, and grow. Whether you're just starting out or scaling up, SB Works is where your journey
begins.
        </p>
        <button onClick={() => navigate('/authenticate')}>Join Now</button>
      </div>
    </div>
  </div>
 )
}
export default Landing;
```

App.js

```
      import './App.css';  // Main app-wide CSS
import { Route, Routes } from 'react-router-dom';  // React Router v6 components
// Common components
import Navbar from './components/Navbar';
// General pages
import Landing from './pages/Landing';
import Authenticate from './pages/Authenticate';
// Freelancer pages
import Freelancer from './pages/freelancer/Freelancer';
import AllProjects from './pages/freelancer/AllProjects';
import MyProjects from './pages/freelancer/MyProjects';
```

```
import MyApplications from './pages/freelancer/MyApplications';
import ProjectData from './pages/freelancer/ProjectData';
// Client pages
import Client from './pages/client/Client';
import ProjectApplications from './pages/client/ProjectApplications';
import NewProject from './pages/client/NewProject';
import ProjectWorking from './pages/client/ProjectWorking';
// Admin pages
import Admin from './pages/admin/Admin';
import AdminProjects from './pages/admin/AdminProjects';
import AllApplications from './pages/admin/AllApplications';
import AllUsers from './pages/admin/AllUsers';

function App() {
  return (
    <div className="App">
      {/* Navbar visible on all routes */}
      <Navbar />
      {/* Define all route paths */}
      <Routes>
        {/* Public routes */}
        <Route exact path='/' element={<Landing />} />
        <Route path='/authenticate' element={<Authenticate />} />
        {/* Freelancer-specific routes */}
        <Route path='/freelancer' element={<Freelancer />} />
        <Route path='/all-projects' element={<AllProjects />} />
        <Route path='/my-projects' element={<MyProjects />} />
        <Route path='/myApplications' element={<MyApplications />} />
        <Route path='/project/:id' element={<ProjectData />} />
        {/* Client-specific routes */}
        <Route path='/client' element={<Client />} />
```

```
        <Route path='/project-applications' element={<ProjectApplications />} />
        <Route path='/new-project' element={<NewProject />} />
        <Route path='/client-project/:id' element={<ProjectWorking />} />
        {/* Admin-specific routes */}
        <Route path='/admin' element={<Admin />} />
        <Route path='/admin-projects' element={<AdminProjects />} />
        <Route path='/admin-applications' element={<AllApplications />} />
        <Route path='/all-users' element={<AllUsers />} />
      </Routes>
    </div>
  );
}
export default App;
```

Server :

Index.js

```
 import express from 'express';
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import bcrypt from 'bcrypt';
import { Application, Chat, Freelancer, Project, User } from './Schema.js';
import { Server } from 'socket.io';
import http from 'http';
import SocketHandler from './SocketHandler.js';
const app = express();
app.use(express.json());
app.use(bodyParser.json({limit: "30mb", extended: true}))
app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
app.use(cors());
```

```
const server = http.createServer(app);
const io = new Server(server, {
   cors: {
      origin: '*',
      methods: ['GET', 'POST', 'PUT', 'DELETE']
   }
});

io.on("connection", (socket) =>{
   console.log("User connected");
   SocketHandler(socket);
})const PORT = 6001;
mongoose.connect('mongodb://localhost:27017/Freelancing',{
   useNewUrlParser: true,
   useUnifiedTopology: true
}).then(()=>{    app.post('/register', async (req, res) =>{
      try{
           const {username, email, password, usertype} = req.body;
         const salt = await bcrypt.genSalt();
          const passwordHash = await bcrypt.hash(password, salt);
          const newUser = new User({
             username,
             email,
             password: passwordHash,
             usertype
          });
          const user = await newUser.save();
          if(usertype === 'freelancer'){
             const newFreelancer = new Freelancer({
                userId: user._id
             })
```

```
            await newFreelancer.save();
        }
          res.status(200).json(user);
        }catch(err){
        res.status(500).json({error: err.message});
    }
});
app.post('/login', async (req, res) =>{
    try{
        const {email, password} = req.body;
        const user = await User.findOne({email:email});
        if(!user) return res.status(400).json({msg: "User does not exist"});

        const isMatch = await bcrypt.compare(password, user.password);
        if(!isMatch) return res.status(400).json({msg: "Invalid credentials"});

        res.status(200).json(user);
    }catch(err){
        res.status(500).json({error: err.message});
    }
});


app.get('/fetch-freelancer/:id', async(req, res)=>{
    try{

        const freelancer = await Freelancer.findOne({userId: req.params.id});

        res.status(200).json(freelancer);

    }catch(err){
```

```
        res.status(500).json({error: err.message});
    }
})

app.post('/update-freelancer', async(req, res)=>{
    const {freelancerId, updateSkills, description} = req.body;
    try{

        const freelancer = await Freelancer.findById(freelancerId);

        let skills = updateSkills.split(',');

        freelancer.skills = skills;
        freelancer.description = description;

        await freelancer.save();

        res.status(200).json(freelancer);

    }catch(err){
        res.status(500).json({error: err.message});
    }
})


// fetch project

app.get('/fetch-project/:id', async(req, res)=>{
    try{

        const project = await Project.findById( req.params.id);
```

```
        res.status(200).json(project);

    }catch(err){
        res.status(500).json({error: err.message});
    }
})

// fetch all projects
app.get('/fetch-projects', async(req, res)=>{
    try{

        const projects = await Project.find();

        res.status(200).json(projects);

    }catch(err){
        res.status(500).json({error: err.message});
    }
})

app.post('/new-project', async(req, res)=>{
    const {title, description, budget, skills, clientId,  clientName,  clientEmail} = req.body;
    try{

        const projectSkills = skills.split(',');
        const newProject = new Project({
            title,
            description,
            budget,
            skills: projectSkills,
```

```
            clientId,
            clientName,
            clientEmail,
            postedDate: new Date()
        })
        await newProject.save();
        res.status(200).json({message: "Project added"});
    }catch(err){
        res.status(500).json({error: err.message});
    }
})
// make bid
app.post('/make-bid', async(req, res)=>{
    const {clientId, freelancerId, projectId, proposal, bidAmount, estimatedTime} = req.body;
    try{
        const freelancer = await User.findById(freelancerId);
        const freelancerData = await Freelancer.findOne({userId: freelancerId});
        const project = await Project.findById(projectId);
        const client = await User.findById(clientId);
        const newApplication = new Application({
            projectId,
            clientId,
            clientName: client.username,
            clientEmail: client.email,
            freelancerId,
            freelancerName: freelancer.username,
            freelancerEmail: freelancer.email,
            freelancerSkills: freelancerData.skills,
            title: project.title,
            description: project.description,
            budget: project.budget,
```

```
            requiredSkills: project.skills,
            proposal,
            bidAmount,
            estimatedTime
        })
        const application = await newApplication.save();
        project.bids.push(freelancerId);
        project.bidAmounts.push(parseInt(bidAmount));
        console.log(application);
        if(application){
            freelancerData.applications.push(application._id)
        }
        await freelancerData.save();
        await project.save();
        res.status(200).json({message: "bidding successful"});
    }catch(err){
        console.log(err)
        res.status(500).json({error: err.message});
    }
})
// fetch all applications
app.get('/fetch-applications', async(req, res)=>{
    try{
        const applications = await Application.find();

        res.status(200).json(applications);
    }catch(err){
        res.status(500).json({error: err.message});
    }
})
// approve application
```

```
app.get('/approve-application/:id', async(req, res)=>{
    try{
        const application = await Application.findById(req.params.id);
        const project = await Project.findById(application.projectId);
        const freelancer = await Freelancer.findOne({userId: application.freelancerId});
        const user = await User.findById(application.freelancerId);
        application.status = 'Accepted';
        await application.save();
        const remainingApplications = await Application.find({projectId: application.projectId, status:
"Pending"});
        remainingApplications.map(async (appli)=>{
            appli.status === 'Rejected';
            await appli.save();
        })
        project.freelancerId = freelancer.userId;
        project.freelancerName = user.email;
        project.budget = application.bidAmount;
        project.status = "Assigned";
        freelancer.currentProjects.push(project._id);
        await project.save();
        await freelancer.save();
        res.status(200).json({message: "Application approved!!"});
    }catch(err){
        console.log(err);
        res.status(500).json({error: err.message});
    }
})
// reject application
app.get('/reject-application/:id', async(req, res)=>{
    try{
        const application = await Application.findById(req.params.id);
```

```
        application.status = 'Rejected';
        await application.save();
        res.status(200).json({message: "Application rejected!!"});
      }catch(err){
        res.status(500).json({error: err.message});
      }
    })
    // submit project
    app.post('/submit-project', async(req, res)=>{
      const {clientId, freelancerId, projectId, projectLink, manualLink, submissionDescription} =
req.body;
      try{
        const project = await Project.findById(projectId);
        project.projectLink = projectLink;
        project.manulaLink = manualLink;
        project.submissionDescription = submissionDescription;
        project.submission = true;
        await project.save();
        await project.save();
        res.status(200).json({message: "Project added"});
      }catch(err){
        res.status(500).json({error: err.message});
      }
    });

   // approve submission
    app.get('/approve-submission/:id', async(req, res)=>{
      try{
        const project = await Project.findById(req.params.id);
        const freelancer = await Freelancer.findOne({userId: project.freelancerId});
        project.submissionAccepted = true;
```

```
        project.status = "Completed";
        freelancer.currentProjects.pop(project._id);
        freelancer.completedProjects.push(project._id);
        freelancer.funds = parseInt(freelancer.funds) + parseInt(project.budget);
        await project.save();
        await freelancer.save();
        res.status(200).json({message: "submission approved"});
    }catch(err){
        res.status(500).json({error: err.message});
    }
});
// reject submission
app.get('/reject-submission/:id', async(req, res)=>{
    try{
        const project = await Project.findById(req.params.id);
        project.submission =  false;
        project.projectLink =  "";
        project.manulaLink =  "";
        project.submissionDescription =  "";
        await project.save();
        res.status(200).json({message: "submission approved"});
    }catch(err){
        res.status(500).json({error: err.message});
    }
});
// fetch all users
app.get('/fetch-users', async(req, res)=>{
    try{
        const users = await User.find();
        res.status(200).json(users);
    }catch(err){
```

```
        res.status(500).json({error: err.message});
      }
   })
   // fetch chats
   app.get('/fetch-chats/:id', async(req, res)=>{
      try{
         const chats = await Chat.findById(req.params.id);
         console.log(chats);
         res.status(200).json(chats);
      }catch(err){
         res.status(500).json({error: err.message});
      }
   })
   server.listen(PORT, ()=>{
      console.log(`Running @ ${PORT}`);
   });
}).catch((e)=> console.log(`Error in db connection ${e}`));
```

Schema.js

```
import mongoose, { Schema, mongo } from "mongoose";

// User schema: Stores general user info (clients or freelancers)
const userSchema = mongoose.Schema({
   username: {
      type: String,
      require: true // Name of the user (required)
   },
   email: {
      type: String,
      require: true, // Email of the user (required)
      unique: true   // Must be unique across users
```

```
  },
  password: {
    type: String,
    require: true // Password for authentication
  },
  usertype: {
    type: String,
    require: true // 'freelancer' or 'client'
  }
});

// Freelancer schema: Additional data for freelancer users
const freelancerSchema = mongoose.Schema({
  userId: String, // Reference to the User document
  skills: {
    type: Array,
    default: [] // Skills offered by freelancer
  },
  description: {
    type: String,
    default: "" // Freelancer profile description
  },
  currentProjects: {
    type: Array,
    default: [] // IDs of ongoing projects
  },
  completedProjects: {
    type: Array,
    default: [] // IDs of completed projects
  },
  applications: {
```

```
    type: Array,
    default: [] // IDs of job applications made
  },
  funds: {
    type: Number,
    default: 0 // Total earnings or wallet balance
  },
});

// Project schema: Stores project postings by clients
const projectSchema = mongoose.Schema({
  clientId: String,          // ID of the client who posted
  clientName: String,        // Client's name
  clientEmail: String,       // Client's email
  title: String,             // Title of the project
  description: String,       // Project details
  budget: Number,            // Estimated project budget
  skills: Array,             // Required skills
  bids: Array,               // Freelancer IDs who have bid
  bidAmounts: Array,         // Corresponding bid amounts
  postedDate: String,        // Date the project was posted
  status: {
    type: String,
    default: "Available"   // "Available", "Ongoing", or "Completed"
  },
  freelancerId: String,      // ID of selected freelancer
  freelancerName: String,    // Name of the selected freelancer
  deadline: String,          // Submission deadline
  submission: {
    type: Boolean,
    default: false         // Has submission been made
```

```
  },
  submissionAccepted: {
    type: Boolean,
    default: false        // Client accepted the submission
  },
  projectLink: {
    type: String,
    default: ""           // Link to completed project files
  },
  manulaLink: {
    type: String,
    default: ""           // Possibly a manual or documentation link (typo? "manualLink"?)
  },
  submissionDescription: {
    type: String,
    default: ""           // Notes submitted with final delivery
  },
});

// Application schema: Stores applications submitted by freelancers
const applicationSchema = mongoose.Schema({
  projectId: String,        // Target project ID
  clientId: String,         // Client who posted the project
  clientName: String,
  clientEmail: String,
  freelancerId: String,     // Applying freelancer's ID
  freelancerName: String,
  freelancerEmail: String,
  freelancerSkills: Array,
  title: String,            // Project title (duplicate info for fast access)
  description: String,      // Project description
```

```
    budget: Number,
    requiredSkills: Array,
    proposal: String,          // Cover letter or proposal message
    bidAmount: Number,         // Freelancer's quoted price
    estimatedTime: Number,     // Estimated days to complete
    status: {
        type: String,
        default: "Pending"     // "Pending", "Accepted", "Rejected"
    }
});

// Chat schema: Stores chat messages between two users
const chatSchema = mongoose.Schema({
    _id: {
        type: String,
        require: true // Custom chat ID (could be user1_user2 format)
    },
    messages: {
        type: Array // List of message objects (e.g., sender, text, time)
    }
});

// Exporting all schemas as Mongoose models
export const User = mongoose.model('users', userSchema);
export const Freelancer = mongoose.model('freelancer', freelancerSchema);
export const Project = mongoose.model('projects', projectSchema);
export const Application = mongoose.model('applications', applicationSchema);
export const Chat = mongoose.model('chats', chatSchema);
```

Dataset Link

This project does not use an external dataset. However, all data is dynamically generated and stored via user input (freelancer profiles, project postings, communication logs, etc.) in a MongoDB database.

GitHub Link : https://github.com/Nayeem000007/apsche-project-main