**Implementation of MERN: A Stack of**

**Technologies to Design Effective Web Based Freelancing Applications**

**Team Members Name:**

1. **HARINI. G**
2. **HARINI. R**
3. **HEMAPRIYA. C**
4. **JOTHILAKSHMI. R**

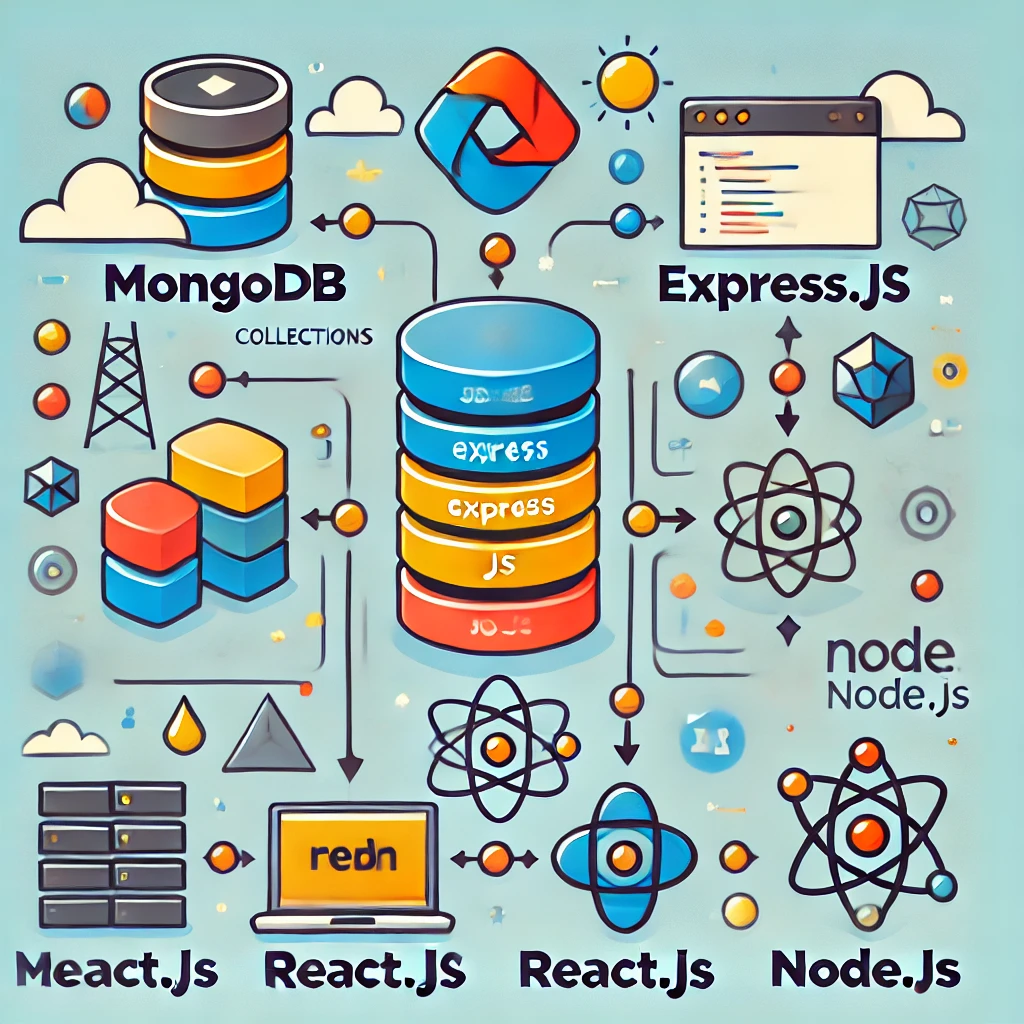
**Institution:** TJS Engineering College.

**ABSTRACT**

In today’s world with the growth of internet, there is a hike in demand for electronically deliverable services. These services can be provided by freelancers. Freelancers are the personals who work individually and provide services as per the need of client according to their contract for a particular job. Every now and then freelancers need a platform where different businesses can be found and contacted. The idea is to have a software that provides the functionality of a hub where both freelancers and business providers can interact with each other. Our work is known as ‘Developer’s Hub’, which encapsulates the idea of freelancing with MERN stack in an optimized way over the web. MERN stack, as the name suggests is stack of four different technologies which are MongoDB, Express.js, React.js, Node.js. This paper reflects significance of freelancing and the technologies concerned in MERN stack. It includes the implementation of MERN for optimizing freelancing over web. The paper also includes challenges faced by freshers or experienced people in freelancing market with the growth in demand of freelancing world

**INTRODUCTION**

As always, every business tries to decrease their overhead and supplement their existing workforce, here is where electronically deliverable services comes into play. Freelancing involves an individual who works with commitment only to particular project and no further involvements. The new generation is more involved into the freelancing as it offers work to freelancers with their own conditions and flexibility of choosing people to work with and along with a choice of projects out of thousands available on different freelancing websites. It allows a vast opportunity of growth and development with strong earning potential for every project chosen. But freelancers face some challenges like they need to find new projects to work on, on frequent intervals. To overcome this, there are multiple websites like Upwork, truelancer.com, guru, freelancing.com which provides both businesses and freelancers multiple options where they can fulfil their requirements. One such website is **Developer's Hub**, where both freelancers and business can find freelancers and project opportunities respectively. A well established and optimized recommendation system is embedded in the project for business category of users by which they get recommendations of freelancers as per their needs on basis of skills. The main purpose of developer’s hub is to help students as current issues faced by students includes difficult to find projects and grow their network. Developer’s hub provides them all these opportunities on a single platform, which is free of cost. It permits them to practice from scratch to expertise level. MERN stack is a promising platform that works on back-end as well as front-end. This stack is composed of four marvellous technologies like MongoDB, React.js, Express.js, Node.js. This is proved to be the best suite to the system as MERN is an open-source JavaScript code which can be used to implement complex system in simplest way and gets improvised with time. Front end part of the system is developed using React.js, which is open source front-end JavaScript framework used to create UI because of it “less code more function” abilities. MongoDB is handling the database of the system as MongoDB is NoSQL database program which are quite useful for working with large sets of distributed data. While Express.js and Node.js is handling the back-end of the website as helps to manage servers and routes.



**THE ARCHITECTURE of MERN STACK**

In MERN Stack the four famous technologies MongoDB, Expess.js, Node.js, and React.js merged are designed to build a robust framework for helping developers to practice with JavaScript components for solving real-life problems and daily development needs. MERN stack is an abbreviation of four different technologies used in the development of dynamic mobile and web applications, where **M** is abbreviated for **MongoDB**, **E** is abbreviated for **Express**, **R** is abbreviated for **React,** and **N** is abbreviated for **Node**. It’s one of the most popular tech stacks used by developers nowadays because it makes the development process easier quicker and smoother. MongoDB is referred to as a document database where data is stored in groups of documents known as collection. Express.js is solely used to create a web server and it is inherited from Node itself therefore it contains all the features of the node with some advancement which makes it more flexible and scalable. React is used to create a modern client-side application. Each technology has become the core of web applications in this modern era.

## A. MongoDB : The Database Layer

MongoDB is a NoSQL database known for its flexibility and scalability. Unlike relational databases that store data in tables with rows and columns, MongoDB uses a document-oriented approach. Data is stored in JSON-like documents (in BSON format), which can include nested fields and arrays, making it a suitable choice for handling complex data structures.

**Key Features of MongoDB:**

1. **Document-Based Structure:**
   * MongoDB stores data in documents, which are structured as key-value pairs.
   * Each document can hold complex, nested data structures, unlike the rigid schema of SQL databases.
2. **Collections and Documents:**
   * A collection is a group of documents. Collections are roughly equivalent to tables in SQL databases.
   * Each document in a collection can have a different structure, making MongoDB schema-flexible. This flexibility allows developers to modify the schema on the go without impacting existing data.
3. **BSON Format:**
   * Data in MongoDB is stored in BSON (Binary JSON), which is an extended version of JSON. BSON allows MongoDB to store data more efficiently by supporting additional data types like Date, which JSON does not natively support.
   * BSON improves the performance and scalability of the database, which is important for handling large-scale applications.
4. **Indexing and Query Optimization:**
   * MongoDB supports complex querying and indexing, allowing for fast access to data. It offers strong query-processing capabilities with support for various query types (e.g., range queries, field queries).
   * Indexes in MongoDB improve search performance and are highly customizable to specific data retrieval needs.
5. **Horizontal Scaling:**
   * MongoDB supports horizontal scaling through sharding, which allows large datasets to be distributed across multiple servers. This scaling capability is critical for applications that manage large amounts of data and experience high traffic.
6. **Security:**
   * MongoDB includes features like authentication, authorization, and encryption at rest, making it a secure choice for managing user and application data.

**Set Up MongoDB with Mongoose (Data Models)**

**1.User model:**

// backend/models/User.js

const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({

name: String,

email: { type: String, unique: true, required: true },

password: { type: String, required: true },

role: { type: String, enum: ["freelancer", "client"], default: "client" },

skills: [String],

bio: String,

});

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

**2.Project model:**

// backend/models/Project.js

const mongoose = require("mongoose");

const projectSchema = new mongoose.Schema({

title: String,

description: String,

budget: Number,

client: { type: mongoose.Schema.Types.ObjectId, ref: "User" },

proposals: [{ type: mongoose.Schema.Types.ObjectId, ref: "Proposal" }],

});

module.exports = mongoose.model("Project", projectSchema);

**3.Proposal** **model:**

// backend/models/Proposal.js

const mongoose = require("mongoose");

const proposalSchema = new mongoose.Schema({

freelancer: { type: mongoose.Schema.Types.ObjectId, ref: "User" },

project: { type: mongoose.Schema.Types.ObjectId, ref: "Project" },

coverLetter: String,

price: Number,

status: { type: String, enum: ["pending", "accepted", "rejected"], default: "pending" },

});

module.exports = mongoose.model("Proposal", proposalSchema);

## B. Express : The Web Application Framework

**Express.js** is a backend web application framework that runs on Node.js and is designed to create RESTful APIs and web applications. It acts as a bridge between the frontend (React) and the backend (MongoDB through Node.js), handling HTTP requests, managing routing, and connecting to databases.

**Key Features of Express.js:**

1. **Simplified Middleware Framework**:
   * Express has a robust **middleware** system that allows developers to handle HTTP requests, apply authentication, and log activities. Middleware functions can process requests at various points in the pipeline, making the application more modular.
2. **Routing**:
   * Express.js offers a simple yet powerful routing system that enables developers to define routes for different endpoints. This system allows for organized and efficient handling of URL requests and responses, making it easier to manage application flows.
3. **Integration with Node.js**:
   * Express inherits all Node.js features, including its event-driven, non-blocking I/O operations, which enhance application performance.
   * It also makes Node’s core capabilities more accessible by abstracting complex code, allowing for a smoother development process.
4. **Scalability**:
   * Express.js is highly scalable due to its minimalistic structure, which requires minimal resources, and because it works seamlessly with Node.js to handle asynchronous requests. This allows the backend to process numerous concurrent requests efficiently.

**API Routes (Express)**

**1.User Routes**:

// backend/routes/userRoutes.js

const express = require("express");

const { registerUser, loginUser, getUserProfile } = require("../controllers/userController");

const router = express.Router();

router.post("/register", registerUser);

router.post("/login", loginUser);

router.get("/profile/:id", getUserProfile);

module.exports = router;

**2.Project Routes**:

// backend/routes/projectRoutes.js

const express = require("express");

const { createProject, getAllProjects, getProjectById } = require("../controllers/projectController");

const router = express.Router();

router.post("/", createProject);

router.get("/", getAllProjects);

router.get("/:id", getProjectById);

module.exports = router;

**Controller Logic:**

**1.User Controller**:

// backend/controllers/userController.js

const User = require("../models/User");

const registerUser = async (req, res) => {

try {

const newUser = new User(req.body);

await newUser.save();

res.status(201).json({ message: "User registered successfully" });

} catch (error) {

res.status(400).json({ error: error.message });

}

};

// Define `loginUser`

### C. React : The Frontend Library

**React.js** is a popular JavaScript library developed by Facebook for building dynamic user interfaces. It is primarily used for creating the **view layer** in web applications, enabling developers to build and manage complex UIs more easily.

**Key Features of React.js:**

1. **Component-Based Architecture**:
   * React uses a component-based architecture where each UI element is broken down into reusable, self-contained components. This modular approach enables developers to maintain a more organized codebase and update UI components independently.
2. **Virtual DOM**:
   * React uses a **Virtual DOM** to improve the performance of rendering changes to the UI. When data changes, React first updates the Virtual DOM, compares it to the actual DOM, and then only updates the elements that have changed. This minimizes browser repaint and reflow, enhancing performance, especially in large applications.
3. **One-Way Data Binding**:
   * React employs a one-way data-binding system where data flows in a single direction, from parent to child components. This approach makes the data flow predictable and easier to debug, allowing for a more stable UI.
4. **Declarative Syntax**:
   * React uses a declarative syntax for defining UI components. This syntax allows developers to describe how the UI should appear at any given time, and React takes care of rendering it accordingly. This results in code that is more readable and easier to maintain.
5. **React Hooks**:
   * React introduced **Hooks** as a way to manage state and lifecycle methods within functional components. Hooks simplify the development process by allowing for more flexible and reusable code, enabling developers to manage complex UIs without using class components.

**1.User Registration Form**

// frontend/src/components/Register.js

import React, { useState } from "react";

import axios from "axios";

const Register = () => {

const [formData, setFormData] = useState({

name: "",

email: "",

password: "",

role: "client",

});

const handleChange = (e) => setFormData({ ...formData, [e.target.name]: e.target.value });

const handleSubmit = async (e) => {

e.preventDefault();

try {

await axios.post("/api/users/register", formData);

alert("Registration successful");

} catch (error) {

alert("Error registering user");

}

};

return (

<form onSubmit={handleSubmit}>

<input name="name" onChange={handleChange} placeholder="Name" />

<input name="email" onChange={handleChange} placeholder="Email" />

<input name="password" type="password" onChange={handleChange} placeholder="Password" />

<select name="role" onChange={handleChange}>

<option value="client">Client</option>

<option value="freelancer">Freelancer</option>

</select>

<button type="submit">Register</button>

</form>

);

};

export default Register;

**2.Project Listing**

// frontend/src/components/ProjectList.js

import React, { useEffect, useState } from "react";

import axios from "axios";

const ProjectList = () => {

const [projects, setProjects] = useState([]);

useEffect(() => {

const fetchProjects = async () => {

const res = await axios.get("/api/projects");

setProjects(res.data);

};

fetchProjects();

}, []);

return (

<div>

<h2>Available Projects</h2>

<ul>

{projects.map((project) => (

<li key={project.\_id}>

<h3>{project.title}</h3>

<p>{project.description}</p>

<p>Budget: ${project.budget}</p>

</li>

))}

</ul>

</div>

);

};

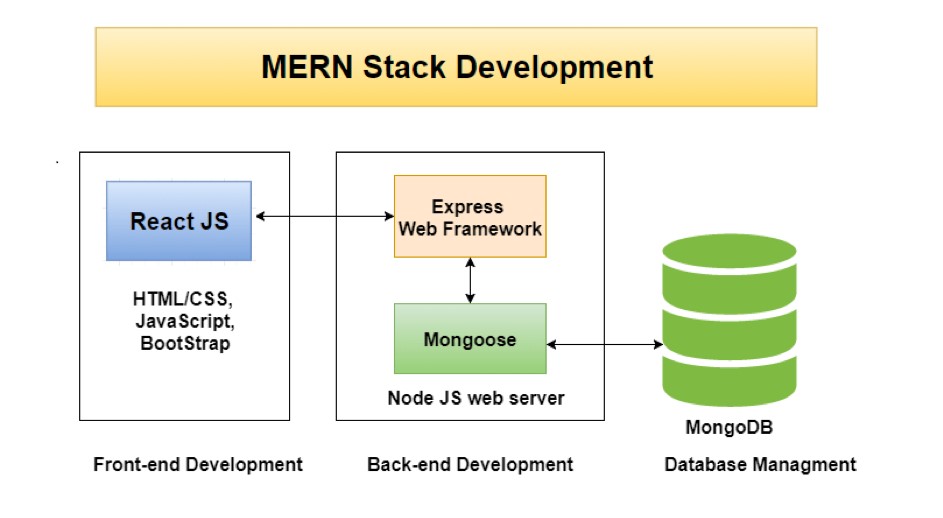
export default ProjectList;

### D. Node : The Runtime Environment

**Node.js** is a JavaScript runtime built on Google’s V8 JavaScript engine. It enables JavaScript to run on the server side, allowing developers to use JavaScript as a full-stack language. Node.js is known for its **non-blocking, event-driven architecture**.

**Key Features of Node.js:**

1. **Event-Driven and Non-Blocking I/O**:
   * Node.js operates on an **event-driven** and **non-blocking I/O** model, making it highly efficient for handling multiple concurrent requests. It can process multiple requests without waiting for previous requests to complete, enhancing application performance.
2. **Single-Threaded Model**:
   * Node.js uses a single-threaded model for handling requests but employs an event loop to manage concurrent connections. This model is particularly efficient for I/O-bound operations, such as file and database interactions.
3. **NPM (Node Package Manager)**:
   * Node.js has an extensive package ecosystem known as **NPM**. With thousands of libraries available, developers can integrate pre-built packages and modules into their applications, reducing development time.
4. **Scalability**:
   * Node.js supports horizontal scaling, which makes it suitable for building large-scale applications. By adding more instances of Node processes, applications can scale to handle a greater number of requests.
5. **JSON Support**:
   * As a JavaScript-based environment, Node.js works seamlessly with JSON, the data format used by MongoDB. This compatibility makes data exchange between the client, server, and database smooth and efficient.



The **Model-View-Controller (MVC)** is coming into existence in 1979 by computer scientist Trygve Mikkjel Heyerdahl Reenskaug. He defined an architectural pattern that divides the complexity of web applications into three logical components: the **model**, the **view**, and the **controller.** Laravel, Ruby on Rails, and Angular are one of the few popular web frameworks which follows MVC architectural pattern. MERN Stack breakdown of MVC pattern:

1. **Model**: Mongoose serves as a ‘Model’ of the MVC pattern at the server-side module. It handles all the data-related logic that the user works with. Mongoose stores all data components of the application that needs to have a function.
2. **View**: React serves as a ‘View’ of the MVC pattern at the client-side module. The module will be written in JavaScript, HTML, and CSS, using ReactJS as the framework. This works as an interface between the user and the features of the application.
3. **Controller**: Express and Node serves as the ‘Controller’ of the MVC pattern. Express and node handle all the functional programming aspects and acts as an interface between Models and Views. It sends the React module to the client’s device and receives HTTP requests from the client and responds according to them.

**Running the Project**

1. **Backend**: Set up the server with express and mongoose, connecting it to MongoDB.

// backend/server.js

const express = require("express");

const mongoose = require("mongoose");

const userRoutes = require("./routes/userRoutes");

const projectRoutes = require("./routes/projectRoutes");

const app = express();

app.use(express.json());

app.use("/api/users", userRoutes);

app.use("/api/projects", projectRoutes);

mongoose.connect("mongodb://localhost:27017/freelancing", {

useNewUrlParser: true,

useUnifiedTopology: true,

}).then(() => console.log("Connected to MongoDB"));

app.listen(5000, () => console.log("Server running on [http://localhost:5000")](http://localhost:5000%22)));

**2.Frontend**: Install dependencies (React, axios) and set up the App.js and index.js files to render components like Register and ProjectList.

// frontend/src/App.js

import React from "react";

import Register from "./components/Register";

import ProjectList from "./components/ProjectList";

function App() {

return (

<div>

<Register />

<ProjectList />

</div>

);

}

**CHALLENGES for MERN STACK**

Web development technologies go through exemplary changes with time. HTML, CSS, and JavaScript have become the cornerstone of web development but no longer fulfil the demands of modern world websites like having better user experience, lesser loading time, and responsiveness to different size devices. Even the MERN stack with the above features is not enough to meet the inevitable expectations of users and developers. Challenges faced by developers while using MERN Stack are:

## MongoDB

**1. Query Challenges (Lack of Join Functionality)**

* **Explanation**: MongoDB is a document-based database, meaning it stores data in collections of documents rather than in rows and tables like a relational database. This document-based structure makes it highly scalable and flexible, but it does not support native join operations as relational databases do.
* **Problem**: In a relational database like MySQL, you can use SQL joins to query across different tables directly. In MongoDB, each query is limited to a single collection (or "table" equivalent), so if you want to retrieve related data from multiple collections, you have to query each collection separately. This can increase complexity and add latency, as multiple queries must be executed and then merged in the application logic.
* **Workaround**: Developers often use embedded documents (storing related data inside the same document) or denormalization (duplicating data across collections) to avoid joins, but these techniques can lead to data inconsistency and larger document sizes. Some recent improvements like the $lookup operator in MongoDB’s aggregation pipeline mimic join behaviour, but it’s still not as flexible or efficient as relational joins in SQL.

**2. Mobile Synchronization Problems**

* **Explanation**: While MongoDB is popular for web applications, mobile applications often require specific support for offline synchronization and data consistency across devices, which MongoDB does not provide out-of-the-box.
* **Problem**: Mobile applications need to sync data between the server and local storage for offline functionality. MongoDB lacks direct support for this type of synchronization, so developers need to build additional logic to ensure data consistency. If users have poor network connectivity, maintaining up-to-date data across all devices becomes even more challenging.
* **Impact**: This lack of built-in mobile support can increase development time and lead to complex code that manages synchronization manually. It can also impact the user experience, as users may need constant connectivity to access updated data reliably.
* **Workaround**: To manage this, developers may use additional tools like MongoDB Realm (a platform by MongoDB that offers synchronization services) or other third-party libraries to sync mobile and server data, but this adds complexity and costs to the application.

**3. Scalability Challenges**

* **Explanation**: MongoDB is designed to be horizontally scalable, meaning it can scale by distributing data across multiple servers (sharding). However, certain performance limitations arise when using MongoDB in a single-node configuration.
* **Problem**: When running MongoDB on a single server, its concurrency is limited because it loads data into memory for fast access. This approach can lead to increased memory usage and slow response times, especially when there are many concurrent users accessing the database simultaneously.
* **Impact**: Applications with high traffic or real-time needs may face performance bottlenecks due to limited concurrency on a single node. As more users try to access or modify data, the server may slow down, affecting the overall responsiveness of the application.
* **Workaround**: To improve scalability, MongoDB can be set up with sharding across multiple nodes, but this requires more infrastructure, setup, and maintenance. This can be costly and complex to implement for smaller applications, though necessary for large-scale applications.

**4. Locking Mechanism**

* **Explanation**: MongoDB uses an optimistic concurrency control approach, where transactions are locked during modification to prevent conflicts. This ensures data integrity, but it has drawbacks.
* **Problem**: When a document is being modified, MongoDB locks it to prevent other sessions from making conflicting changes. If multiple transactions attempt to modify the same document, MongoDB will abort and retry conflicting transactions, which can lead to unpredictable order of requests and a nondeterministic execution sequence.
* **Impact**: In a high-traffic application, this approach can result in delays, as aborted transactions must wait to retry, potentially slowing down the application and creating a poor user experience. For applications with complex transactions, the lack of fine-grained locking can limit efficiency.
* **Workaround**: MongoDB 4.0 introduced multi-document ACID transactions, which improve data consistency and help manage some of these issues, though they can still impact performance in high-concurrency environments.

## React

1. **Complexity of React Components in Complex Applications**

* **Explanation**: React applications are built using components, which are reusable UI elements. While this approach works well for simpler projects, creating and managing components in large, complex applications can be challenging.
* **Problem**: As the number of components grows, managing state and props (data passed between components) becomes more complicated. Components can become tightly coupled or deeply nested, which makes debugging and modifying specific parts of the application difficult. Additionally, managing component lifecycle methods, state updates, and side effects (e.g., network requests or timers) can be intricate in large applications.
* **Impact**: This complexity can slow down development, as developers may struggle to understand how data flows through nested components. As a result, maintaining and scaling a React application requires careful planning and often a strong understanding of best practices.
* **Solution**: Libraries like **Redux** and **React Context** can help manage global state, reducing the complexity of prop drilling (passing data through multiple component layers). However, these solutions add additional learning curves and dependencies.

1. **Precision Limitations of the React Virtual DOM**

* **Explanation**: The Virtual DOM (VDOM) in React is a JavaScript-based representation of the actual DOM. React uses the VDOM to track changes and efficiently update only the parts of the real DOM that need to change. This improves performance for many applications, especially when compared to directly manipulating the DOM.
* **Problem**: The VDOM is not always precise. When there are frequent or complex updates, the React VDOM can lead to performance bottlenecks, as it still has to compare new virtual DOM trees with previous ones (a process called "reconciliation"). This can be particularly noticeable in applications with complex animations, large datasets, or frequent state changes, where even minor inefficiencies are amplified.
* **Impact**: Applications with frequent state updates, animations, or extensive interactivity may experience slower rendering performance, impacting the user experience.
* **Solution**: Optimizing rendering by using React.memo, PureComponent, and other React hooks like useMemo and useCallback can help manage rendering performance. In complex applications, server-side rendering (SSR) or using frameworks like **Next.js** can improve speed, but these add more layers to the tech stack.

1. **Complicated Asynchronous Programming for Server Interaction**

* **Explanation**: React applications often need to fetch and display data from a server, which requires asynchronous programming. JavaScript's asynchronous features (like Promises and async/await) allow data

fetching without freezing the user interface.

* **Problem**: While async/await syntax has made asynchronous code more readable, managing multiple asynchronous tasks (such as fetching data, handling responses, or managing complex chains of requests) can still be challenging. Handling errors, retries, and managing loading states across different components adds to the complexity. Developers must also account for scenarios like data caching, throttling, or syncing, which can be intricate to manage.
* **Impact**: In complex applications, poorly managed asynchronous code can lead to bugs, inconsistent data, or a poor user experience due to frequent reloads or loading states.
* **Solution**: Libraries like **Axios** (for making HTTP requests) and **React Query** (for data fetching and caching) help manage asynchronous operations and improve code readability. These libraries simplify error handling, caching, and server data synchronization but add additional dependencies and learning requirements.

1. **Limitations of HTML Templates in React**

* **Explanation**: React does not use traditional HTML templates. Instead, it uses **JSX**, a syntax extension that allows HTML-like syntax to be written within JavaScript code. JSX provides a more programmatic approach to rendering UI elements than traditional HTML templates.
* **Problem**: While JSX can be powerful, it is not a complete HTML templating system. For example, certain HTML features like conditional attributes or custom data-binding mechanisms are not natively supported. Developers have to use JavaScript expressions to handle conditions and loops within JSX, which can become cumbersome and make JSX code harder to read.
* **Impact**: In complex components with a lot of conditional rendering, JSX can get hard to maintain and read compared to HTML templates in other frameworks (like **Vue.js** or **Angular**). This can slow down development, as developers spend more time managing the JSX syntax for rendering logic.
* **Solution**: Structuring components with clear separation of concerns and using helper functions to handle complex logic can help improve readability. React also allows for custom hooks and utilities that can manage complex rendering, but this requires careful organization and adherence to best practices.

1. **Performance Issues on Low-Spec Devices**

* **Explanation**: React's component-based and VDOM-based architecture is generally optimized for performance, but as the application grows, it can become memory-intensive and processor-demanding.
* **Problem**: On low-spec devices (such as entry-level laptops, older smartphones, or tablets), React’s VDOM reconciliation process, component re-renders, and memory usage can cause slowdowns, especially if the application has large data sets or complex animations. Additionally, some React features like useEffect and useState may trigger unnecessary re-renders if not carefully managed.
* **Impact**: Users on low-spec devices may experience a laggy interface, increased loading times, and slower response rates, negatively impacting the overall user experience.
* **Solution**: To improve performance, developers can optimize component re-rendering using React.memo, useMemo, and useCallback to prevent unnecessary updates. Breaking down large components, using lazy loading for assets, and compressing resources can also help. Tools like **React Profiler** are useful for identifying performance bottlenecks, allowing developers to optimize specific components for lower-spec devices.

1. **Node**
2. **Lack of Standardized Coding Rules and Guidelines**

**Explanation**:

Node.js is designed with flexibility in mind, allowing developers to structure and write their code as they see fit. Unlike other frameworks or languages that have strict conventions or opinionated architectures, Node.js doesn’t impose a rigid structure.

**Problem:**

This flexibility can lead to inconsistency, especially for beginners or teams working on larger projects. New developers or those unfamiliar with Node.js may find it difficult to know where to start or what practices to follow, as there isn’t a universally accepted “Node.js way” to structure code. This means that each project may have a different architecture and coding style, making it harder to maintain and scale.

# Impact:

Lack of standard guidelines can lead to unorganized code and “spaghetti code,” where functionality and logic are spread in an inconsistent manner. This not only affects readability but also makes it challenging for teams to collaborate, onboard new members, and maintain a consistent codebase. It can also lead to errors if the code doesn’t follow a well-thought-out structure.

# Solution:

To address this, teams often create their own coding guidelines and use patterns like MVC (Model-View-Controller) for structuring applications. Using linters (e.g., ESLint) and formatters, along with popular style guides (e.g., Airbnb or StandardJS), can also help enforce consistency. However, setting up these practices requires extra effort and is often a learning curve for new developers.

**2.Poor Documentation and Fragmented Community Practices**

# Explanation:

Documentation is crucial for learning and using any technology effectively. Since Node.js is open-source and has a large, diverse community, its documentation and best practices can be inconsistent and sometimes sparse, especially for community-created modules and libraries.

# Problem:

# While official Node.js documentation exists, many third-party libraries and tools lack comprehensive or beginner-friendly documentation. This can make it hard for developers to find and understand the resources they need. Additionally, since Node.js is constantly evolving, some documentation may be outdated, further complicating the learning process. The open-source nature of Node.js means there are multiple solutions to the same problem, which can be confusing and overwhelming.

# Impact:

# Poor documentation can slow down development, as developers spend more time troubleshooting and researching solutions. It also creates a steep learning curve for beginners, who may find it difficult to connect different components of Node.js applications or implement best practices. This can result in errors or inefficient solutions, as developers may not fully understand the available tools or patterns.

# Solution:

# To work around documentation gaps, developers rely heavily on community forums, Q&A sites (like Stack Overflow), tutorials, and guides. Using established libraries with strong community support (like Express.js for routing) can also improve the development experience. However, these resources may vary in quality, and it’s often up to the developer to validate and test different approaches, which can be time-consuming.

# DEVELOPER’S HUB: A MERN STACK WEB APPLICATION

Developer’s Hub is a full-stack web application developed using MongoDB, Express.js, React.js, and Node.js. The objective of the application is to develop an environment that can help freelancers whether he or she is an experienced individuals or a student by providing them with hundreds of projects on a single platform with just a few tap aways. The main aim of the project was to provide students an opportunity to start their careers in the freelancing world without any mandatory need for a degree or any other certificates. By evaluating similar applications and articles it can be easily found that when a freelancer struggles to find a job, they must conquer multiple turbulence to reach out to a single project owner. Such turbulence is bidding processes, with irrelevant tests and costly courses to be completed before entering some communities. This application will showcase suitable freelancers for every job posted based on the skill requirements of business owners. To develop this recommendation system, the application consists of a content-based recommendation system that works on a real-time basis. The intended user of the applications is freelancers, Businesses, or any recruiters and an admin who can manage data at MongoDB databases. Following are the user stories explaining how the application works for Freelancers and Businesses.

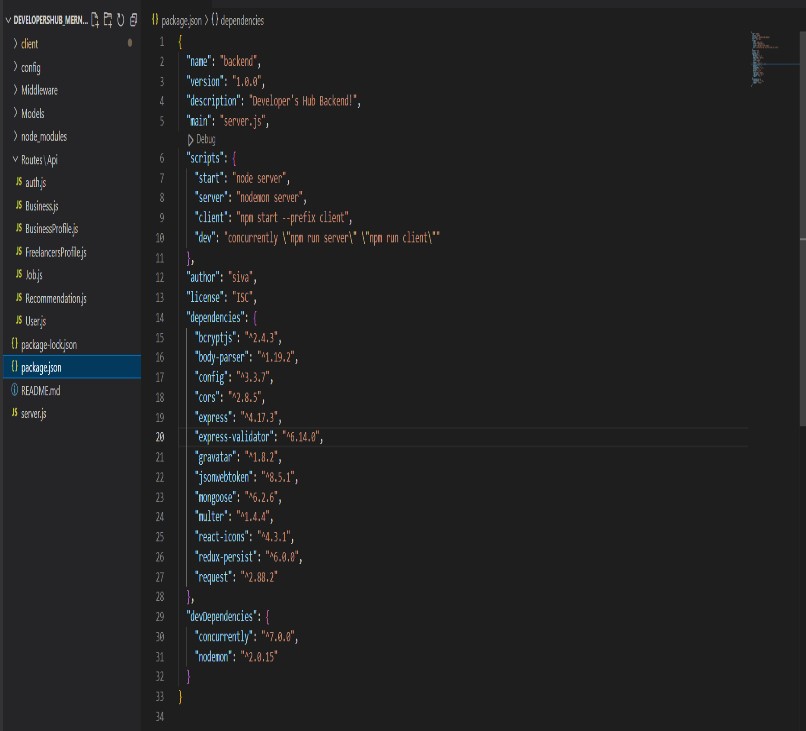
To have all the privileges of the application any freelancer who has skills can register themselves on the web app and create their profile. The application allows users to view or save necessary details like contact numbers, email ids, social links, etc. It also allows freelancers to mark their interests in any project posted by businesses. Once a user creates their profile, it can be seen in the freelancer’s list by other freelancers and businesses. Businesses can also see the freelancer’s profile when they mark interest in the jobs posted by them. An updating feature is also implemented in the app to add new experiences and education details. To communicate with the job posters their email has been provided on their profile. Businesses can also register themselves and create their profile. For businesses, the application allows entering details like contact numbers, email ids, company links, etc. They also have access to create a job posting and mention the required skills and salary range they wanted it to be. After creating these job posts they can see a list of interested freelancers, recommended freelancers, and a list of all freelancers at any time. If required, they can contact to the freelancer with the email id provided for them.

# IMPLEMENTATION

As with every other MERN Stack application Developer’s hub is also divided into sections i.e., clientside and server-side.

## A. Server-side Implementation

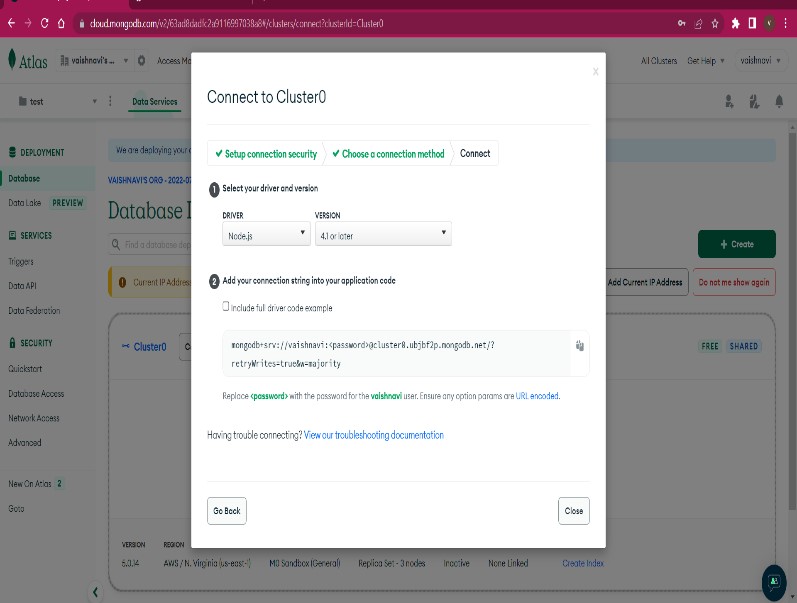
The server side of the application is implemented by using built-in modules or customized modules of Node.js.Node.js has more than one million packages to install various frameworks or development tools used to develop the project. To install npm packages, execute “npm install” command in the command line interface. Such dependencies installed on the developer’s hub application are jsonwebtokens, reacticons, mongoose etc. Another part of server-side implementation is database management which is executed with MongoDB. That’s why server-side implementation is further divided into sections which are database management, Routes, middleware, Models, and server setup and configuration.



**Package.json**

**1)Database Management:**

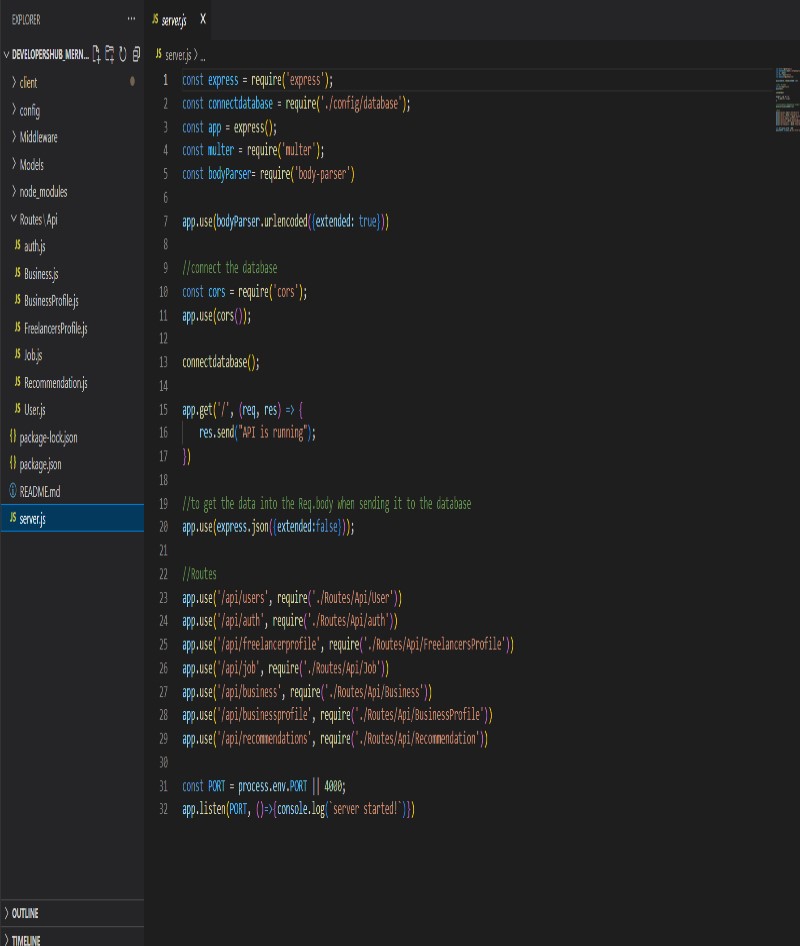
MongoDB utilizes data by storing them in form of clusters. Clusters handle large volumes of data and the number of requests. It makes monitoring and automation easy through data reduction and load balancing. To set up the cluster of this project create a cluster by setting up the cloud provider and region. MongoDB also provides backup and size options for data at very reasonable prices. After creating the cluster, connect the database and application by pasting a link into the config folder of the project. This connection allows live updates and modification of data done by the client directly to the database. Atlas admin can also monitor or set conditions if required to the database. In this application, default conditions are perfectly fine with the requirements.



# Connect cluster

**2) Server Setup:**

Start connecting the server with express.js, cors, multer, and body-parser with require() in the server.js file to include its external node modules. To connect the server on visiting to a particular route on any browser app.use() is implemented with two parameters that will download the customized react components on accessing them.



**Server.js**

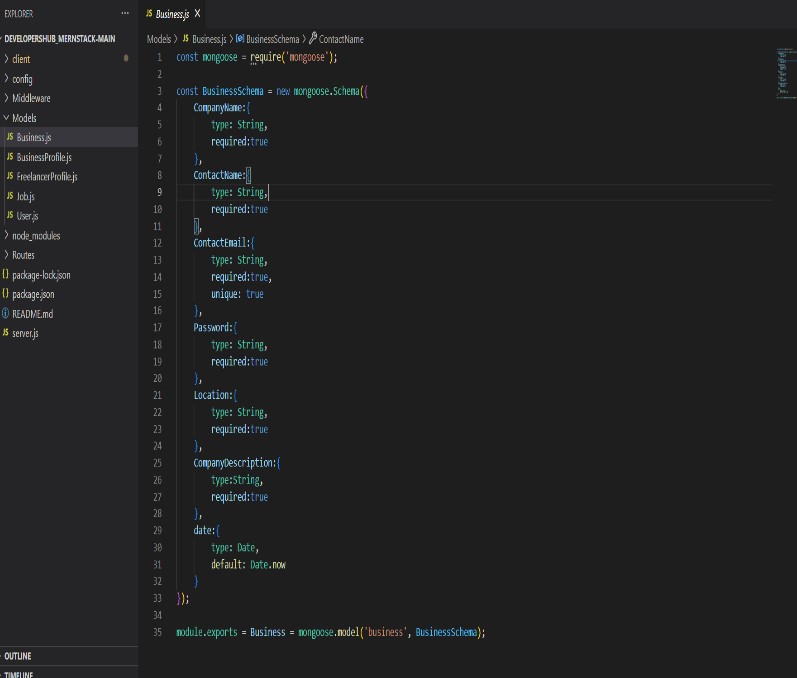
# 

**3) Models:**

Once the database and server are set, to define the structure of the application start creating schemas. The mongoose models which define the application’s instances, and their properties are Business.js, BusinessProfile.js, FreelancerProfile.js, Jobs.js, User.js. Every property which has the required method as “true” is mandatory to carry out operations while if it is “false” it is optional. Business.js and user.js models are customized to register in the application by businessman and freelancer respectively while BusinessProfile.js

and FreelancerProfile.js is customized to create profiles. In job.js,

“mongoose.Schema.Types.ObjectID” is implementing a very fine feature of the application. It is connecting the jobs with the interested freelancer defined under the user.js model.

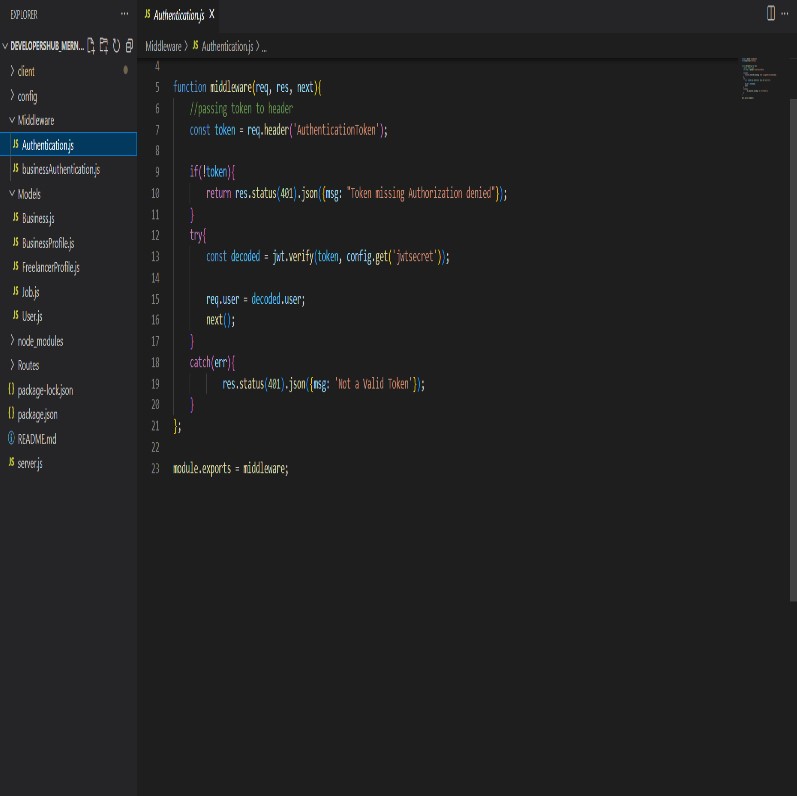


# Business.js

The password property defined in these models is not stored in form of plain text in the MongoDB database. It is first hashed using bcrypt which is a very popular npm module. gensalt() function in bcrypt is used to implement the hashing of the password. This hashed password is further encrypted by the hash () function and send to the mongodb. If the password is needed in any sign in or sign-up activity of the application, an authentication process is carried out in the web application. For security purposes Json web token authentication level of the application. Whenever a user register on the website a token is generated using jwt.sign() and verified each time anyone log in to the website using jwt.verify().

1. **Middleware:**

Middleware in this application is the authentication files that contain objects and functions which is executed during the request and response cycle of the application. Req is known as the request object, Res is known as the response object and next is used for the next middleware function. Middleware works with the JWT authentication system to secure the web application. JWT can be installed using the command “npm install jsonwebtoken” on the command line interface. Whenever a user enters login details, this dependency creates a web token and returns it to the browser. ‘AuthenticationToken’ is created as middleware to verify the token. It is sent to the authentication header and jet.verify() matches the data entered by user to verify and allow user to get the response.



# Authentication.js

1. **Routes**:

The interface of the web application is defined using the routes. It connects the request made on the client side to the correct backend resources. Four different types of HTTP methods are used in the routes are post(): to send data to the server, get(): to request data from various resources, delete(): to remove the data from the database, put(): to update the data. The routes customized while implementing the applications are:

User.js: Implements post method while freelancers register themselves on the application. In the registration process, a web token is generated, and the entered details are posted on the server.

Business.js: Implements post method while businesses register themselves on the application. In the registration process, a web token is generated, and the entered details are posted on the server.

Auth.js: Implements post method when a user enters username and password for signing into the server which then generates jwt token. It also implements a get method that authenticates the user by jwt token and sends the user details to the client side.

FreelancerProfile.js: It is implemented to get or delete the details of all freelancers or any specific freelancer details by using their id from the server. It is also used as a method of updating experience and education details.

BusinessProfile.js: It is implemented to get the details of all businesses or any specific business details by using their id from the server. It also implements a post method which is used to create and update the profile details of a registered business.

Job.js: It is implemented to post or delete jobs by businesses and freelancers can get all of them or by any specific id. Freelance can put or delete their interests in various jobs.

Recommendation.js: Implements get method by showing freelancers which could be suitable for jobs on basis of the skills a freelancer

## B. Client-side Implementation

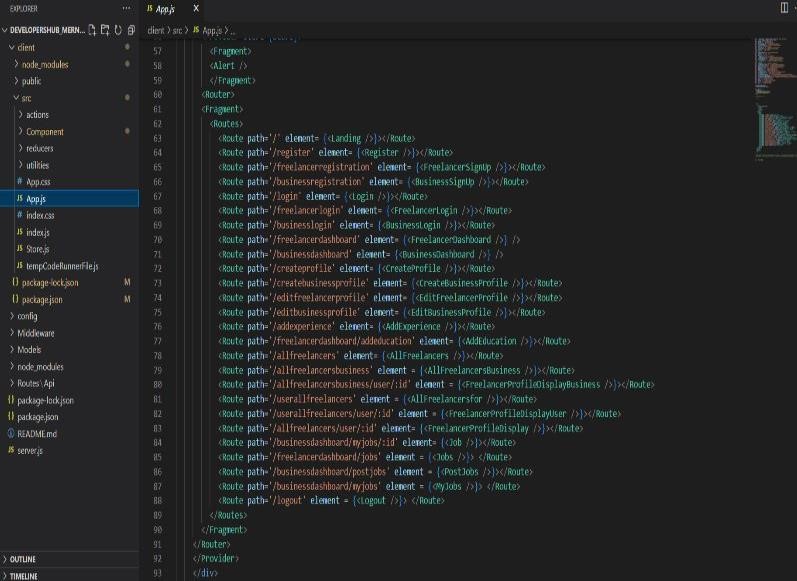
The project's client side is implemented using only React.js using third-party libraries from npm. React.js optimized the output by allowing live editing in CSSand JavaScript in the early phases of the project. The “npx create-react-app app-name” command is used to set up the development environment. Here, npx is a react tool that is used to run packages. The packages installed throughout the projects are “moment”, “React redux “, and “React bootstrap”. The moment is used to format the dates like when a freelancer applied for a particular job.

1. **React Components:**

React components are the building blocks of this application. React has built-in components of independent codes which could be reused in order to make the whole code data reductant and easy to update whenever needed. Every page in the application whether it is the landing page, create a profile page, or the page to find jobs is built on the react components. Every page has its own DOM structure.For instance, the Landing page uses <carousel> component to create the banner. Under this <carousel> tag it uses <carousel.caption>, <carousel.item> to add heading and images on the banner.

1. **React-router-dom:**

In order to gain security only authenticated users are allowed to visit any page of the application. If any person whether a freelancer or businessman, the app forces that person follows a particular route to have access to any feature. For this purpose, **react-router-dom** is installed next in the process. To have access to the routes, all components of a particular react file needs to be rendered using the <Route> tag between <Routes> which is updated with version 6 of react-router-dom.



# Routes

1. **React redux:**

React redux is an openly sourced JavaScript library that is used in this application for its capability of managing states with a unidirectional flow of data models of the MERN Stack applications. Redux works as a virtual cloud on the client side of the application as it manages multiple states at a time. It reads and updates data for multiple components for making the UI interface easy to render. For instance, Jobs posted by some businesses got stored in the redux cloud which is also known as the redux store. This can be executed with help of Axios by writing a code of actions.

1. **Actions**:

Actions are objects which contain a blank field for the user to fill the information with some event that happened in the application. Action implanted in this application are:

Alert: It is implemented to create a dialogue box and alert the users whenever needed.

Authentication: It is implemented for both freelancers and businesses to login into their profiles and allows them to access routes that are protected. Only a user which is registered before can access these routes.

Profile: It is also implemented for both freelancers and businesses to create or update their profiles. In the case of freelancers, they can use defined functions to add experience and education while in the case of businesses they can add jobs and other details of their business.

Job: It is implemented for business users to create them by adding different details of the job and an add intrested() method is used by them to view interested freelancers. There are defined reducers for each action created. Reducers are methods that take the state of the components and actions taken as their parameter and return a new state.

***REAL-WORLD USE:***

*In a real-world application, especially for a platform that connects freelancers and clients, several critical features enhance functionality, security, and user experience. Here’s a detailed explanation of some key components you would encounter:*

***1. Authentication (e.g., JWT)***

***Overview:*** *Authentication is crucial in any web application that manages user accounts and sensitive data. JWT (JSON Web Token) is a widely used authentication standard that provides secure, stateless user authentication, allowing users to log in and access protected resources.*

***How JWT Works:***

* *Token Creation: When a user logs in, the server verifies the credentials. Upon successful verification, the server generates a JWT containing user information (e.g., user ID, roles) and signs it with a secret key.*
* *Token Structure: A JWT consists of three parts: the header, payload, and signature. The payload contains encoded user information, and the signature ensures the token’s integrity.*
* *Token Storage: The token is then sent to the client, often stored in local storage or cookies.*
* *Token Validation: For subsequent requests, the client includes the token in the request header. The server checks the token’s signature to authenticate the user without needing session storage, making JWT stateless.*

***Benefits:***

* *Stateless: JWTs don’t require server-side session storage, which improves scalability.*
* *Security: Since JWTs are signed and can be encrypted, they provide a secure way of transmitting user credentials.*
* *Flexibility: JWTs can be used across domains, making them useful for microservices or distributed applications.*

***Challenges:***

* *Token Expiry and Refreshing: JWTs need expiration times to avoid unauthorized access if a token is stolen. Implementing refresh tokens can ensure tokens are updated securely.*
* *Storage and Security: Securely storing tokens on the client side is crucial to prevent XSS (cross-site scripting) attacks.*

***2. Payment Processing (e.g., Stripe, PayPal)***

***Overview:*** *Integrating payment processing allows users to pay and receive payments through secure, third-party payment gateways. Stripe and PayPal are popular options due to their extensive features, security, and ease of integration*

*.*

***How Payment Processing Works:***

* *Integration: Using APIs provided by Stripe or PayPal, you can implement features such as creating charges, managing subscriptions, and handling refunds.*
* *Secure Checkout: Users are redirected to a secure, hosted payment page provided by the gateway, or payments can be handled directly on the site using tokens and secure protocols (e.g., HTTPS, PCI compliance).*
* *Tokenization: Payment providers use tokenization, where sensitive card details are converted into tokens, making it possible to securely process payments without storing sensitive information on the server.*
* *Transaction Handling: Once a payment is completed, the payment provider notifies the server, updating the order status, generating invoices, or handling subscriptions.*

***Benefits:***

* *Security Compliance: Payment providers handle complex PCI DSS (Payment Card Industry Data Security Standard) compliance, ensuring secure transactions without requiring the app to manage sensitive data directly.*
* *Reliability: Payment providers offer high availability and support for multiple payment methods (credit cards, digital wallets, etc.), improving the user experience.*
* *Additional Features: Payment providers like Stripe and PayPal offer advanced features like fraud detection, analytics, and automated invoicing.*

***Challenges:***

* *Transaction Fees: Payment providers charge fees for each transaction, which may impact profits, especially for micro-transactions.*
* *Integration Complexity: While these providers offer SDKs and documentation, integrating with the platform may involve handling webhooks, managing errors, and ensuring data privacy.*

***3. Messaging Between Clients and Freelancers***

***Overview:*** *Messaging is essential in platforms that connect freelancers with clients, as it allows for real-time communication, negotiation, and task coordination. A secure and responsive messaging system improves the user experience and promotes engagement.*

***How Messaging Works:***

* *Real-time Communication: Messaging typically requires real-time features, often implemented using WebSockets or Socket.IO in a Node.js environment. This allows messages to be sent and received instantly without refreshing the page.*
* *Database Management: Messages are stored in a database (e.g., MongoDB) to maintain chat histories, which can be referenced by both users. Messages may include timestamps, status indicators (e.g., read/unread), and file attachments.*
* *User Interface: The frontend needs a chat UI with features like typing indicators, message timestamps, and scrolling history.*
* *Notifications: Notifications (via email, SMS, or in-app notifications) alert users about new messages, helping to keep both parties engaged.*

***Benefits:***

* *Engagement: Messaging allows direct, convenient interaction between users, improving the likelihood of successful transactions.*
* *Real-time Support: Freelancers and clients can resolve issues or questions promptly, enhancing trust and cooperation.*
* *Documented Conversations: Message history allows users to refer back to previous conversations for clarification on project requirements, avoiding miscommunications.*

***Challenges:***

* *Data Security: Messaging systems need to secure messages, especially in platforms dealing with confidential or sensitive information.*
* *Scalability: Handling thousands of concurrent messages requires careful infrastructure design and database management to ensure consistent, fast performance.*
* *Notification Management: Implementing an effective notification system to alert users of new messages can add complexity, especially when handling multiple notification channels (e.g., email, push notifications).*

***4. More Advanced Error Handling and Input Validation***

***Overview:*** *Robust error handling and input validation are essential to maintain security, stability, and usability. Proper error handling ensures that the system can respond gracefully to unexpected issues, while input validation protects against malicious inputs.*

***Error Handling:***

* *Types of Errors: Errors can occur at various levels (e.g., network errors, database errors, user errors, validation errors). Each type should be handled appropriately to avoid crashes or data corruption.*
* *Centralized Error Handling: Implementing a centralized error-handling system (e.g., using middleware in Express.js) allows you to handle errors consistently across the application. Logging and tracking errors help developers identify and fix issues faster.*
* *User-Friendly Error Messages: For user-facing errors, messages should be meaningful and actionable, letting users know how to resolve issues or what went wrong without exposing technical details.*

***Benefits:***

* *Improved Stability: Proper error handling prevents crashes, ensuring that the application can handle unexpected situations.*
* *Enhanced Security: Consistent error handling protects the system from revealing sensitive information through error messages, reducing vulnerability to attacks.*

***Input Validation:***

* *Importance: Input validation is essential for security and data integrity, as it prevents malicious data from being processed by the system. It’s a critical defence against common attacks like SQL injection, NoSQL injection, and cross-site scripting (XSS).*
* *Client and Server Validation: Validation should occur both on the client side (for immediate feedback) and server side (for security). This ensures that inputs conform to expected formats and requirements (e.g., email formats, required fields).*
* *Libraries and Tools: Libraries like Joi (for validation in Node.js) or Validator.js (for data sanitization) simplify input validation. For real-time applications, form libraries like Formik with Yup validation can enhance the user experience.*

***Benefits:***

* *Data Integrity: Ensures that only valid data is processed, reducing the likelihood of application crashes or corrupt data.*
* *Security: Guards against various forms of injection attacks and other malicious inputs, securing the application.*

***Challenges:***

* *Error Tracking and Logging: Implementing a robust error-tracking system can be complex, especially in distributed systems where errors may occur in multiple services.*
* *User Experience: Balancing thorough validation with usability is essential, as overly strict or excessive validation can frustrate users.*

# CHALLENGES FACED

Developing a full-stack application with the MERN stack involves numerous challenges, especially for a developer learning new technologies to create a complete, dynamic, and secure web application. Here’s a detailed look at the challenges faced at each stage:

**1. Learning React.js for a Multipage, Dynamic Frontend**

React.js is a powerful library for building user interfaces, but it requires understanding some complex concepts, especially when building a multipage, dynamic frontend.

* **Challenge**: Managing Components and State
  + React’s component-based architecture means the UI is broken into small, reusable parts. For large applications, managing multiple components, keeping them organized, and managing their individual states can be complex and time-consuming.
* **Challenge**: Handling Routing in a Single-Page Application (SPA)
  + Creating a multipage experience within a single-page application requires using libraries like **React Router**. Learning how to set up routes, nested routes, and redirects is essential to give users a seamless navigation experience, which can be challenging when new to React.
* **Challenge**: Integrating Asynchronous Data Loading
  + Since React is primarily a frontend library, integrating it with an API to fetch data asynchronously involves understanding promises, asynchronous programming, and handling data loading states. This is crucial for dynamic content but can be difficult to get right initially.

**2. Learning React Redux for State Management Across the Application**

As applications grow, managing state across multiple components becomes challenging, especially if several components need access to the same data. Redux is a popular choice for managing global state in React, but it has its own learning curve.

* **Challenge**: Understanding the Redux Flow (Actions, Reducers, and Store)
  + Redux introduces concepts like **actions**, **reducers**, and a **centralized store**. These are different from React’s basic state handling, and understanding how data flows from the store through actions and reducers is crucial. For beginners, this can feel complex and repetitive.
* **Challenge**: Boilerplate Code
  + Redux often requires a lot of setup code for each new feature or piece of state. Setting up actions, reducers, and connecting components to the store can be tedious and require careful management of code to avoid duplication and ensure consistency.
* **Challenge**: Performance and Debugging
  + When state updates frequently or Redux is used incorrectly, it can slow down the app. Efficiently using selectors, optimizing state structure, and debugging with tools like Redux DevTools are necessary to ensure good performance but require extra learning and practice.

**3. Managing and Connecting Frontend and Backend Code**

Connecting a React frontend with a Node.js and Express backend involves handling multiple technologies and managing complex code interactions.

* **Challenge**: Setting Up a Consistent Data Flow
  + Ensuring that data flows smoothly between the client and server requires handling HTTP requests, setting up API routes, and designing the data structure in a way that is compatible with both MongoDB (backend) and React (frontend).
* **Challenge**: Error Handling and Communication Between Components
  + Handling errors across the frontend and backend is essential for a good user experience. This includes understanding how to catch errors, send appropriate error messages from the backend to the frontend, and display error messages in a user-friendly way on the UI.
* **Challenge**: Managing CORS Issues
  + When connecting a frontend to a backend on different servers (such as in development), **Cross-Origin Resource Sharing (CORS)** issues can occur. Configuring CORS middleware on the server to allow requests from the frontend is necessary but can be confusing initially.

**4. Learning Node.js and Express.js for Backend Development**

Node.js and Express.js are crucial for handling the backend server and creating API endpoints, but they introduce unique challenges.

* **Challenge**: Understanding Asynchronous Programming
  + Node.js uses an asynchronous, event-driven model, which can be difficult to learn for developers accustomed to synchronous programming. Handling callbacks, promises, and async/await functions is necessary to manage data flow without blocking server operations.
* **Challenge**: Creating and Structuring API Endpoints
  + Building a RESTful API requires designing endpoints in a logical and organized way, defining routes and methods (e.g., GET, POST, PUT, DELETE), and ensuring that they correctly handle requests and responses. Structuring the server code to manage these endpoints efficiently can be challenging.
* **Challenge**: Managing Middleware for Authentication, Validation, and Error Handling
  + Express.js uses middleware to handle tasks like authentication, input validation, and error handling. Learning how to use and configure middleware effectively is essential for a secure and well-functioning API.

**5. Learning MongoDB as an Alternative to SQL Databases**

Using MongoDB, a NoSQL database, presents its own set of challenges, especially for those accustomed to traditional relational databases.

* **Challenge**: Adapting to Document-Based Data Storage\*\*
  + MongoDB stores data in collections and documents, which is different from tables and rows in SQL. Understanding how to structure data in a schema-less environment and working with **embedded documents** and **arrays** can be challenging.
* **Challenge**: Querying and Aggregation\*\*
  + MongoDB's query syntax is different from SQL’s. Learning MongoDB's **query language** and using the aggregation pipeline to filter, sort, and process data requires practice, especially for complex data queries.
* **Challenge**: Ensuring Data Security and Accessibility\*\*
  + MongoDB offers flexibility, but it requires configuring permissions, authentication, and backup strategies to ensure data security. Learning MongoDB's **Atlas** platform or configuring security settings on a self-hosted database adds an extra layer of complexity.

**6. Learning React-Bootstrap and Material UI for an Eye-Catching Interface**

UI libraries like **React-Bootstrap** and **Material UI** provide pre-designed components that help create visually appealing interfaces, but they come with their own learning curves.

* **Challenge**: Customizing Pre-Designed Components
  + React-Bootstrap and Material UI offer customizable components, but making them fit a specific design aesthetic or application theme requires understanding of CSS, component styling, and the customization API of each library.
* **Challenge**: Managing Component Dependencies
  + Using multiple UI libraries (like both Bootstrap and Material UI) can lead to conflicts in styling or component behavior. Developers need to carefully manage and integrate these dependencies to avoid inconsistencies and ensure a cohesive UI.
* **Challenge**: Mobile Responsiveness
  + Ensuring that the UI looks good on various devices requires understanding how to use responsive utilities and layout components provided by these libraries. Learning how to create a fluid, responsive UI that adjusts to different screen sizes can be time-consuming.

**7. Following Modern Security Measures and Middleware**

Building a secure application requires implementing various security practices and using middleware to protect against common threats.

* **Challenge**: Implementing Authentication and Authorization
  + Securing the application requires implementing authentication mechanisms like JWT (JSON Web Tokens) and ensuring that sensitive data is encrypted. Understanding how to handle tokens on the client side and protect routes on the server side can be complex.
* **Challenge**: Data Validation and Sanitization
  + Input validation and sanitization are crucial to prevent attacks such as SQL injection, even though MongoDB is generally safer from SQL-based attacks. Using libraries like **Joi** for validation and middleware to sanitize user input can add complexity to the development process.
* **Challenge**: Securing API Endpoints
  + Securing API endpoints involves configuring rate limiting, enabling CORS correctly, and using HTTPS. Learning these techniques and applying them consistently requires an understanding of web security best practices.

## CONCLUSION

In conclusion, despite the existence of multiple established freelancing platforms, many entry-level freelancers still face obstacles, from difficult onboarding processes to challenges in building visibility and finding opportunities. These limitations highlight the need for a platform that addresses the unique requirements of newcomers.

The Developer's Hub is designed to open a new, accessible path for freelancers by focusing on user-friendly features, a supportive community, and an intuitive interface. Developed using the MERN Stack (MongoDB, Express.js, React.js, and Node.js), the platform leverages the stack's strengths to create an efficient, scalable, and responsive application.

**Key Takeaways on the MERN Stack’s Role in Development**

* **Data Management with MongoDB**: MongoDB’s non-relational structure allows flexible and scalable data storage. As a NoSQL database, it is well-suited for handling diverse data structures, enabling real-time updates for managing client profiles, job listings, messaging, and transaction data without complex schema constraints.
* **Frontend Interface with React.js**: React's component-based structure is ideal for developing a responsive, interactive frontend that adapts to the needs of both freelancers and clients. React’s virtual DOM ensures efficient rendering, making it easier to build a smooth and engaging user experience across devices.
* **Backend and API Layer with Express.js and Node.js**: Express.js and Node.js power the backend, creating an effective bridge between the frontend and database. Together, they handle HTTP requests, manage authentication, and control real-time data flow between clients and the server. Node.js's event-driven architecture is optimized for handling asynchronous I/O operations, essential for real-time notifications, chat, and other interactive features.

The MERN Stack’s adaptability and efficiency made it possible to develop this platform in a streamlined way, with each technology playing a critical role in supporting the platform's core functionalities. MongoDB, React, Express, and Node together offer a cohesive environment, from managing data to creating an intuitive user interface. With this stack, Developer's Hub can handle real-time user interactions and facilitate data-driven features essential to building a seamless, modern freelancing platform.