UNIVERSITY OF LEICESTER

Department of Informatics University of Leicester

Individual Project (CO7201)

Final Report

Job Recommender System for Freelancers

Name: Siva Kumar Pasupuleti

Email: Skp36@student.le.ac.uk

Student ID: 209021476

Project Supervisor: Yuan, Bo

Second Marker: Radwan, Marwan

Date of Submission – 20-05-2022

Word Count: 12800

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do these amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Siva Kumar Pasupuleti

Date: May 19, 2022

Table of Content

Chapter – 1

Introduction

Motivation

To develop a web application that helps freelancers to work on real-time projects. The additional motivation is to develop a platform where students can demonstrate their skill sets by working on real-time projects.

Aim

Develop a Web application named "Job Recommender System for Freelancers". The purpose of the application is to develop a platform that helps businesses to find freelancers. For example, this application can help small businesses who are trying to develop a website for their customers, it can help businesses who are trying to hire developers based on their technology requirements, and it can help students to work on real-time projects.

Challenges

- Research and Learn MERN stack for web application development. Which includes:
    - Learning React.js for developing the application's front end.
    - Learning and Implementing React Redux (State Management)
    - Develop a user-friendly application.
    - Learning Node.js and Express.js for developing API endpoints.
    - Learning how to use MongoDB database
    - Learning React-Bootstrap for designing the application's UI (User Interface).
- Creating a vast application which involves three different users.
- Following the best security measures for protecting business-related information.

User Story

Freelancers

- Freelancers register with the application, they can either be individuals who are self-employed or students.
- Once they are registered, they will create their profile by providing the necessary details such as the skillset, contact details, portfolio, social media profiles etc.
- Once the profile is created it can be viewed by businesses who have registered within the application.
- Freelancers can edit their profile details whenever necessary. This will help them to add additional skills as they acquire them.
- Freelancers can also view other freelancers who have profiles within the application.
- Freelancers can view the jobs posted by businesses and express their interest if they prefer to work on the job.
- Freelancers can communicate with businesses.

Business

- Businesses can register within the application.
- Once Registered they will be able to create the profile to provide additional information about the business.
- Once the profile creation is complete, the business will be able to post jobs based on their requirements and view freelancers that are available to work.
- These jobs will be available for freelancers to view and express interest.
- Businesses can view their individual jobs and the list of freelancers who express their interest in the job.
- View recommended freelancers for each job. The application generates a list of recommended freelancers who perfectly match the required skill sets of a job.
- If interested in freelancers, they can communicate with them using email.

Who are freelancers in our application?

A person who is self-employed and has the skill set to work on real-time projects. Our application also includes students as freelancers, the purpose is to provide a platform for students to work on real-time projects.

Who are businesses in our application?

Companies who are trying to outsource their real-time projects to Freelancers.

Chapter 2

Background Research

There has been solid background research done on how other applications like this are performing and how to bring in those additional features which will add uniqueness to our application. The background research is segregated into three parts one is based on the unique features that can be brought in, the other research is based on which technology stack to be used and the final research is based on the recommendation system.

Research on finding the best stack for application development

What is a technology stack?

A Technology stack is a combination of frameworks, tools and programming languages that are used in application development. (Horiachko, 2021)

Famous stack's which are used for application development nowadays are LAMP, Python-Django, MEAN, and MERN. (Horiachko, 2021)

Summarizing each stack:

- LAMP – Here, Linux works as the operating system, Apache is the HTTP server, MYSQL is the database and finally PHP which is the server-side programming. (Horiachko, 2021)
- Python-Django – This stack is based on python programming language, for back-end development Django framework is used. Apache is the HTTP server and MYSQL is for database. (Horiachko, 2021)
- MEAN – MongoDB as database, Angular as front end, Node and *ExpressJs* for server-side programming. (Horiachko, 2021)
- MERN – Very efficient stack in developing single page applications, MongoDB as database, *ReactJs* for designing the front-end components, Express and NodeJS for server-side programming. (Horiachko, 2021)

Which stack for development?

After going through the technical requirements for each stack, the author has some experience using *ReactJs* and *NodeJS*. So, MERN Stack would be an ideal choice for developing the application. The advantages of using the MERN stack are further explained in the design chapter of this paper.

Research on similar web applications

There are several applications on the market that help freelancers to sell their work online and help them in finding real-time projects. Some example applications are namely UpWork, Fiverr, and Freelancers. The author has gone through each application and has summarized the features that the applications bring in for helping freelancers. The concern in this application includes the bidding process that each freelancer must go through to get the project and team registrations are encouraged by these applications.

Unique Features in the application

- There are no team registrations allowed and freelancers do not bid on projects.
- Business decides on the budget, duration, and other aspects of the project.
- Providing an opportunity for students to work as freelancers and gain experience of working on a real time project.
- Building a recommendation system to help businesses find freelancers who exactly match their skill set requirements.

Research on how to implement a recommendation engine

A Recommendation system should show the most applicable item to the user. The challenge involved in developing a recommendation system is to understand the requirements of the user and provide suggestions based on the requirements. Providing valid recommendations in real time is the challenge of building a recommendation system. (Anam Khan, 2020)

There are different approaches to building a recommendation engine namely, content-based recommendation engine and collaborative-based filtering. (Anam Khan, 2020)

Collaborative Based filtering

The collaborative recommendation system assumes that if a group of people liked a particular item in the past they will also like a similar item in the future. There are various approaches involved in collaborative filtering such as, "Person who likes this also likes" which is known as the user-user approach, "Since you liked this item before you might also like the following items" which is known as the item-item approach, there are other approaches such as the user-item approach which generates a recommendation based on the combination of both the approaches. (Rocca, 2019)

There are diverse types of collaborative filtering methods such as Memory based, and Model based. All the above-mentioned approaches such as the user-user and item-item approach are related to the memory based collaborative filtering. Model based collaborative approach assumes a latent model which relies on user-item interactions. (Rocca, 2019)

Disadvantages of collaborative filtering

The main disadvantage in collaborative filtering is the cold-start problem, which means without feedback there will be no recommendations made by the system. In our application scenario, we are trying to implement recommendations for business based on the freelancers' skills set. If the freelancer has not had any interaction with the business, he/she might not show up on the recommendation list. (Rocca, 2019) To overcome this disadvantage and to justify the requirements of the application, the author is using a Content-based filtering approach.

Content - Based Filtering

In content-based filtering the recommendations are based on the contents of the user requirements. By following this approach, we can recommend freelancers to business based on the skill set of the freelancer (Rocca, 2019). The idea of the application is to recommend freelancers to jobs based on their skill sets. When the business post job, they can view the recommended freelancers who perfectly match their skill requirements. This is achieved using React redux. The implementation of the content-based filtering is explained in chapter 6 of this paper.

Chapter 3

Requirements

The Requirements have been changed since the beginning of the project based on the supervisor suggestions and timeline. The finalized requirements have been mentioned below.

Essential Requirements

These are the essential requirements that will be implemented into the application. The requirements are categorized based on the users.

Business Providers

- Register
- Post a Job
- View the list of jobs (New, in progress, and completed)

- Get recommended freelancers for the job
- View the list of freelancers available
- View the list of freelancers who showed interest in the job posted
- Confirm the job with the preferred freelancer
- Communicate with freelancers

Freelancers

- Register as a Freelancer
- Create and manage profile
- View the list of available jobs
- If interested in the job, express your interest in working on the job
- Communicating with business providers
- Get notified if selected from the recommended list.

Admin

- View and Manage Jobs
- View and Manage Freelancer/Developers

Recommended Requirements

- Freelancers rating – Skill based.
- Freelancers Portfolio – Defines the skill level in more detail.

Optional Requirements

- Repository to upload job-related files.
- In-app payment feature.
- Communication within freelancers.

Due to the time limitations, the author has implemented all the essential requirements and recommended requirements. The author will be implementing the optional requirements in future work.

Chapter 4

Design

In this Design chapter, the author will be explaining the concepts which are involved in the MERN stack. The author will also be summarizing the advantages and the compatibilities.

Client Server Architecture

This Project follows the client-server architecture. The application running on the web browser acts as the client for both the freelancer and the business. When considering the administrator of the application, the MongoDB cluster which shows the database collection acts as the client. The

application server runs on NodeJS, and the author uses express web framework to write the API end routes. MongoDB is used for storing data.

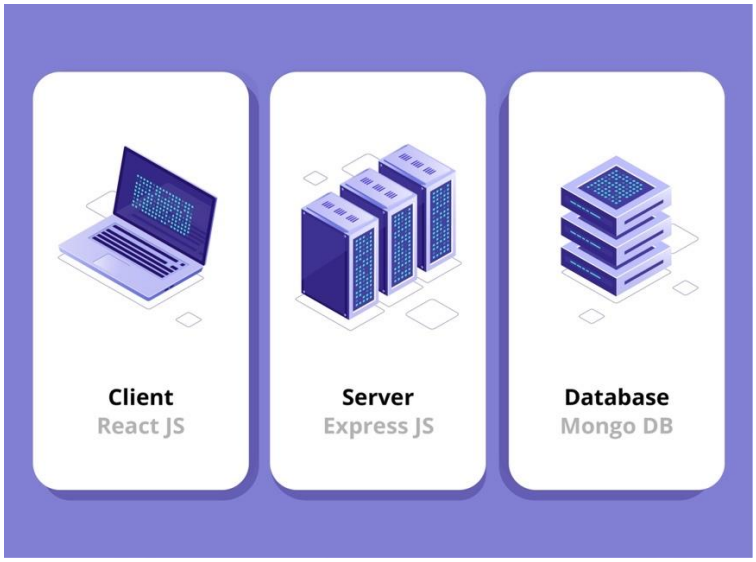


Figure [1]

The MERN Stack

Web development by tradition has been carried out by technologies such as ASP.NET, PHP, or Java servlets. These technologies have a wider community for support and are used and supported by web developers across the world. The problem with Technologies such as the ASP.NET and Java servlets is the limitations and performance issues that they have. (Nguyen, 2020)

Advantages of MERN Stack

- Simple and
- Uniform

MERN stack is a JavaScript based development process. MERN stack involves the use of open-source elements.

- MongoDB as the Database system.
- Express acts as the framework for server.
- To run JavaScript on server node.js is used.
- React.Js acts as the client library.

APIs (application program interfaces) are created using express and node. These APIs are used for logic and database (MongoDB) interactions. React acts as the client and sends the HTTP-based requests to the backend server. The server which is in place will analyze the request made by the react client, based on the request the servers extract data and information from the database and send it back to the react client.

Back-End Development

NodeJS

NodeJS is an open-source run time environment which is cross-platform and can be used for the development of server-side programming and can also be useful for applications networking. The programming language used to write NodeJS is JavaScript using which the development of web applications can be simplified. Nodejs is the core element of the MERN stack. It is a widely used web server environment and is open-sourced, it executes JavaScript code on the server. (Points, n.d.)

NodeJS consists of non-blocking APIs, which means the server will not keep waiting for any data. It keeps moving on to the next available API and will return to the previous API call when the data becomes available. There is a mechanism which notifies when the data is ready. (Points, n.d.)

The core of chrome browser is based on the google V8 engine and NodeJS is dependent on this engine which enhances the applications memory consumption and speed of performance.

Node engine is

- Asynchronous
- Non-blocking and
- Event driven

Nodejs and MongoDB are technologies that are asynchronous. To elaborate, when using NodeJS application can call a request and can move on to the next request instead of awaiting a response for the initial request. When the sent request is successful, the call-back function comes into play by informing the application about the results. This eliminates the need for awaiting a response and supports multiple responses at a time. (Nguyen, 2020)

To summarize we can compare the request handling feature in PHP and Node.js.

How PHP handles requests:

- Send request to a file system
- Wait until the file is open and read
- Return the response to the client
- Move on to the next request

On the other hand, we can see how Node.js handles requests:

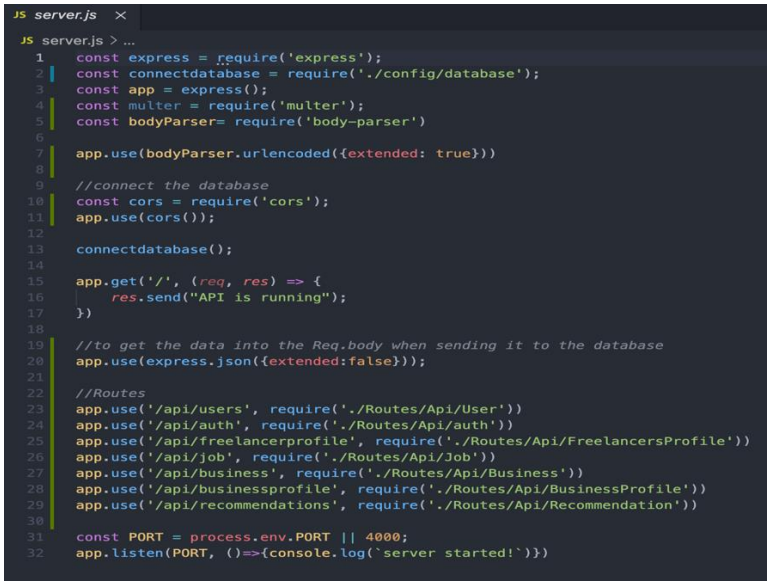
- Send request to the file system
- Move on to the next request
- When the request is complete, the call back function sends the results of the request to the client (Terminates the waiting time)

Node Modules

Node has several built-in modules which can be installed and used directly, they are known as node modules. Node modules are like libraries in JavaScript. They can be installed as a package of functions and can be used within the application whenever necessary. (Nguyen, 2020)

Node Modules features:

- Creating or customizing your own modules.
- Modules that are built-in and can be used without the need for performing installations.
- Express.js is one amongst the best node.js packages which is used nowadays.



```
JS server.js
JS server.js > ...
1   const express = require('express');
2   const connectdatabase = require('./config/database');
3   const app = express();
4   const multer = require('multer');
5   const bodyParser= require('body-parser')
6
7   app.use(bodyParser.urlencoded({extended: true}))
8
9   //connect the database
10  const cors = require('cors');
11  app.use(cors());
12
13  connectdatabase();
14
15  app.get('/', (req, res) => {
16      res.send("API is running");
17  })
18
19  //to get the data into the Req.body when sending it to the database
20  app.use(express.json({extended:false}));
21
22  //Routes
23  app.use('/api/users', require('./Routes/Api/User'))
24  app.use('/api/auth', require('./Routes/Api/auth'))
25  app.use('/api/freelancerprofile', require('./Routes/Api/FreelancersProfile'))
26  app.use('/api/job', require('./Routes/Api/Job'))
27  app.use('/api/business', require('./Routes/Api/Business'))
28  app.use('/api/businessprofile', require('./Routes/Api/BusinessProfile'))
29  app.use('/api/recommendations', require('./Routes/Api/Recommendation'))
30
31  const PORT = process.env.PORT || 4000;
32  app.listen(PORT, ()=>{console.log(`server started!`)})
```

Figure [2]. *Application's Node.js code to create the server*

Node Package Manager (NPM)

Installation and usage of node modules are achieved through the node package manager (NPM).

Advantages of using NPM:

- Version management and package installation through command line interface (CLI)
- NPM registry has more than 1 million open-source packages.

Installation of node packages can be carried out in two ways either globally or locally with the help of the "*npm install*" command.

In our application the node modules are used with the help of the *require ()* function.

```
JS server.js
1   const express = require('express');
2   const connectdatabase = require('./config/database');
3   const app = express();
4   const multer = require('multer');
5   const bodyParser= require('body-parser')
6
```

Figure [3]. *Using node modules*

Package.json

Package.json file will be in the JSON format, which includes the name and version of the package which are installed.

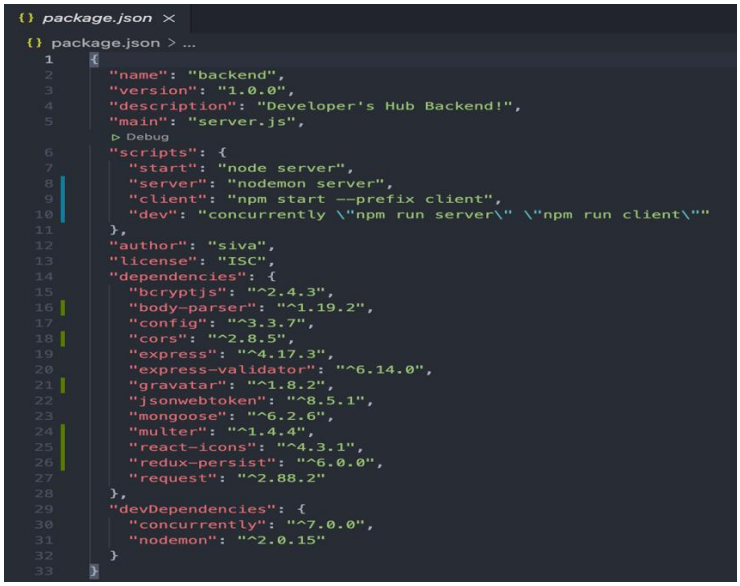


Figure [4]. *Package.JSON file*

As you can see from the above figure, there are quite a few node packages that have been installed into the application such as the *bcryptjs*, *react-icons*, *Multer*, and *gravatar* etc.

Node Framework – Express.js

NodeJS when used solely for server-side programming will not be an asset. This is the reason for working with frameworks such as the *ExpressJS*. (Mozilla, n.d.)

ExpressJS is a framework of NodeJS, and the framework is for backend web application development. *ExpressJS* is open source and free to use. Express is a valuable and efficient framework to develop API endpoints/routes. Some of the features are the routing is made robust, and when using express we can say that high performance is on focus. (Mozilla, n.d.)

Express is a framework based on node, so it includes all the features of node such as:

- Performance
- Scalability
- Flexibility and
- Simplicity

Since express is minimum, developers across the globe have programmed middleware packages which help in solving the issues we might face in web development. Express helps in.

- Robust API creation
- Middleware
- HTTP Methods

Routing

Every application developed has endpoints, routing is a mechanism by which the client requests are routed to the code which can handle them. The express syntax consists of an instance which is an object, to manage the GET and POST requests. (Nguyen, 2020)

Example Routing methods in express:

- App.get ()
- App.post ()
- App.put ()
- App.delete ()

When there is more than one call-back function, we can use *next ()* which is part of the express router. This helps us to switch to the next call-back function in the route.

Middleware

In the Application's request-response cycle, middleware functions have access to req and res and next.

- Req – Request Object
- Res – Response Object

In express, middleware can be programmed and used in diverse types. The types are namely.

- Built-in middleware
  - *express.static*
  - *express.json*
  - *Express.urlencoded*
- Router-level middleware
  - *router.use()*
  - *router.method()* (example methods: Get, Post, Put)
- Third-party middleware
  - Example: *const {check, validationResult} = require('express-validator')*
- Error handling middleware
  - Middleware which is used for handling errors
- Application-level middleware
  - *App.use()*
  - *App.method()* (example methods: Get, Post, Put)

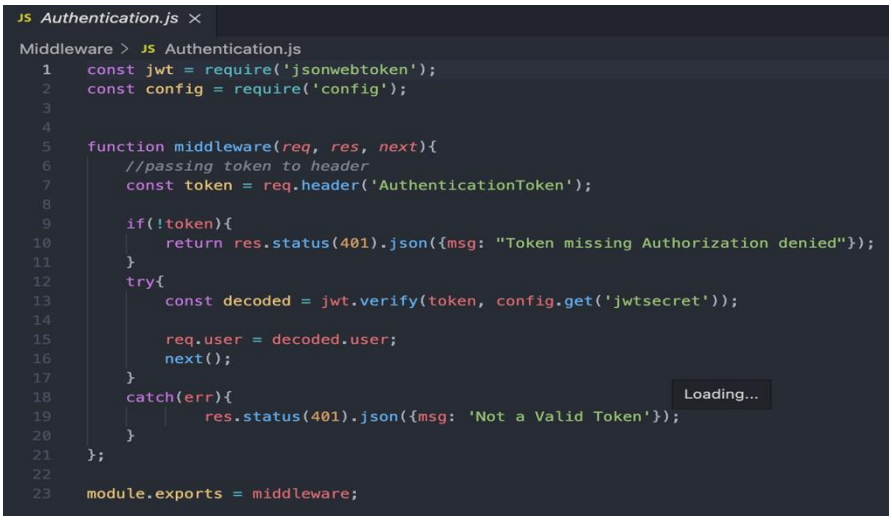


Figure [5]. *Use of Middleware to authenticate users*

Database Management

The Author is using MongoDB as the database system for the application.

MongoDB

MongoDB is a database program which is document oriented. It is a NOSQL database system. MongoDB documents are stored in JSON Format. (Ado Kukic, 2021)

No SQL Databases

There is difference in the way NoSQL database store data when compared to how relational databases store data. Examples of NoSQL database are

- Document databases
- Key-value databases
- Wide-column databases
- Graph databases

The common characteristics between these types are

- Dynamic schema
- Large data loads
- User volume

There are various features in MongoDB, when queried it returns documents with the required fields, we can also use JavaScript functions defined by the user. Random Samples of results can be returned for a specified size. Indexing can be done both primary and secondary. Replicas can be performed with the help of MongoDB. The replicas are maintained as primary and secondary. (Ado Kukic, 2021)

Data Models

A data model is an abstract model, that determines how the data elements are related to each other and how to organize them.

There are two types of data models

- Embedded Data Model
    - The data which is related is put together into a single structure.
- Normalized Data Model
    - Using references to handle the data relations. The references will be embedded into a document, and it will be referenced into other documents to relate amongst them.

Mongoose Model

The logical structure of a database is determined by data model. A Model's Instance is known as document. NodeJS based Object Data Modelling (ODM) library for MongoDB is known as Mongoose. Mongoose aims developers at the application layer to implement a specific schema. Model validation is another feature that the mongoose provides. Mongoose also supports JSON schema validation. (Ado Kukic, 2021)

Advantage of using a NOSQL database helps us from not getting constrained by a rigid data model. (Points, n.d.)

Advantages of using the Mongoose Model:

- Connects MongoDB and Node.js
- Straight-forward
- Schema based
- Data relations control
- Validation of schema
- Allows us to perform database actions(Read, Write, Delete and Update)

Front – End Development

This section explains the technologies and the design features involved in developing the front end of the application

About React

We need a front-end development environment for making client requests to the back end. In this project I will be using React as the front-end library for building the user interfaces. React divides the user interface into various components, using components can also minimize the bugs that are expected to occur when the UI is being built by the developer. With the help of react we can break the code into pieces of components which can be used whenever necessary. When combining these components together we can create a full UI and many of the rendering work is abstracted from the view by react. (Docs, n.d.)

In React the rules are not strictly enforced when it comes to organizing the files and the convention of code. (Docs, n.d.)

The Modern JavaScript Features are utilized by react but react also supports the use of JSX syntax. JSX is nothing but an extension of JavaScript syntax and they are known as JSX expressions. (Docs, n.d.)

In Our Project we have used React for creating the front-end components and the state is managed between the applications components with the help of React-Redux state management. (Docs, n.d.)

Advantages of using React

- JSX
- Virtual DOM

JSX

React uses JSX, which adds to the advantage of using this javascript library. We write react elements and template handling using JSX. JSX is not a mandatory part of using react, but it simplifies the way component code is structured. (Nguyen, 2020)

Benefits of using JSX

- Time Efficiency
- Templating is faster
- Use of javascript code inside the curly braces {}

Virtual DOM

DOM object representation is known as virtual DOM. DOM is responsible for executing all the requests before the UI gets rendered to the user. So, the number of updates in the web page gets complicated. Virtual DOM only updates the differences which reduces the number of updates happening to the UI. This process of checking difference and updating the page only when there is change is the reason for developers choosing react for designing there client side of the application.

Components

In React, the components are based on two types.

- Functional Component and
- Class Component

After certain recent updates in react, there is no need for class component. We can just use components which are based on functions. In our application the components are function based not class based.

React Bootstrap

Replacement of bootstrap JavaScript is React-Bootstrap. The advantage of using React-Bootstrap is eliminating the need for dependencies such as jQuery. Every component when using React-Bootstrap has been built as a true react component. (Bootstrap, n.d.)

It is also one of the oldest libraries, and the growth and evolution of react-bootstrap happened along with react. It is also known to be a UI foundation of excellent choice. (Bootstrap, n.d.)

There are hundreds and thousands of themes which can be used. When using react bootstrap there is no need to add or bring in dependencies. We can use the design elements directly as components and eliminate the need to import styles. (Bootstrap, n.d.)

The Model is components based, so control is given to us over the functions and forms of each component when using the react component model. When implementing the accessibility is kept in mind. (Bootstrap, n.d.)

React Redux

Application state is managed by an open-source JavaScript library known as the React-Redux. With the help of Redux, react builds the user Interface. Redux working methodology involves using store, actions, and reducers. The data is updated when the actions are dispatched to the redux store. (Points, n.d.)

React uses a unidirectional data flow model which assists apps to manage state and scale in a sensible way. The concept of React Redux is simple, but as the size of the application grows, managing states between components can end up being challenging. Redux comes from the flux architecture and additionally eliminates the drawback of the flux. For example, flux consists of many stores whereas redux consists of a single store. (Points, n.d.)

Architecture

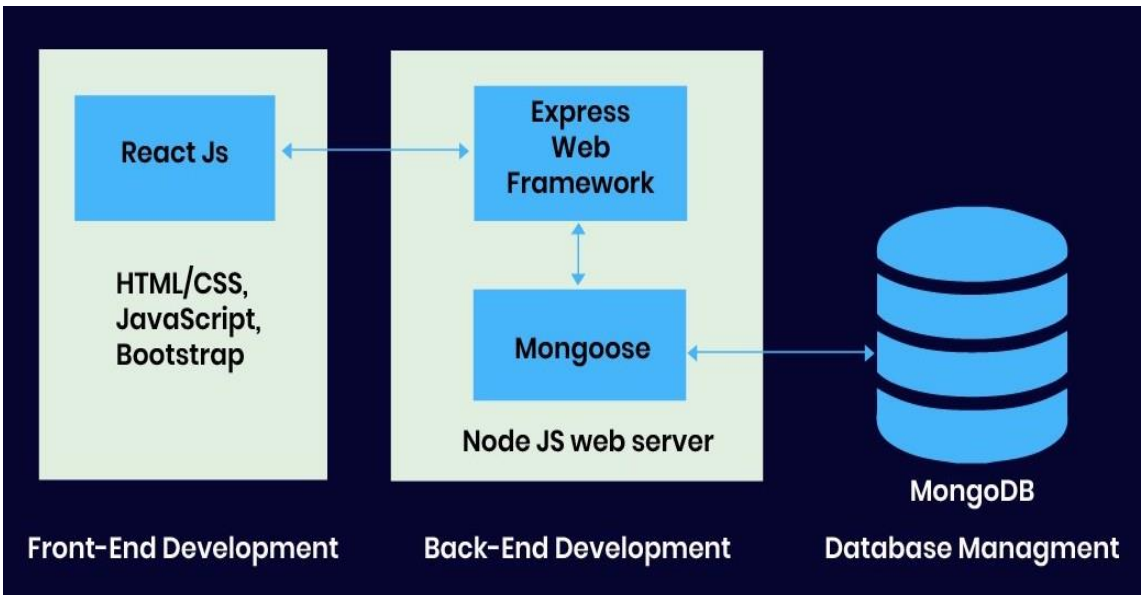


Figure [6]. *Architecture*

Use Case Diagrams

Freelancer Use Case Diagram

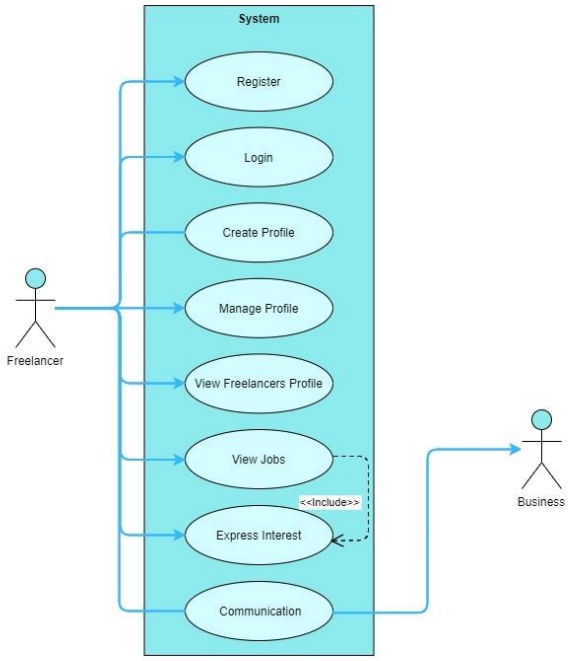


Figure [7]. *Freelancer UseCase*

Business Use Case Diagram

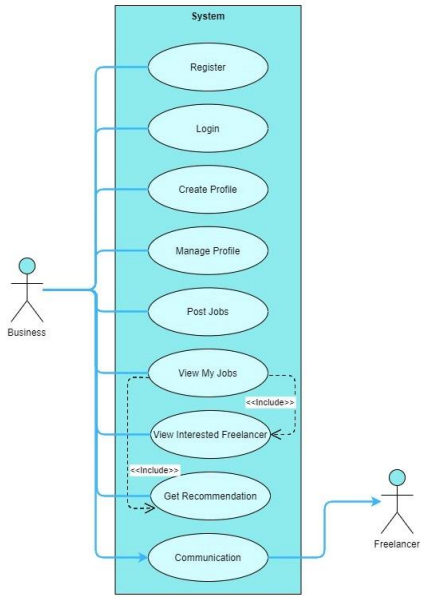


Figure [8]. *Business UseCase*

Sequence Diagram

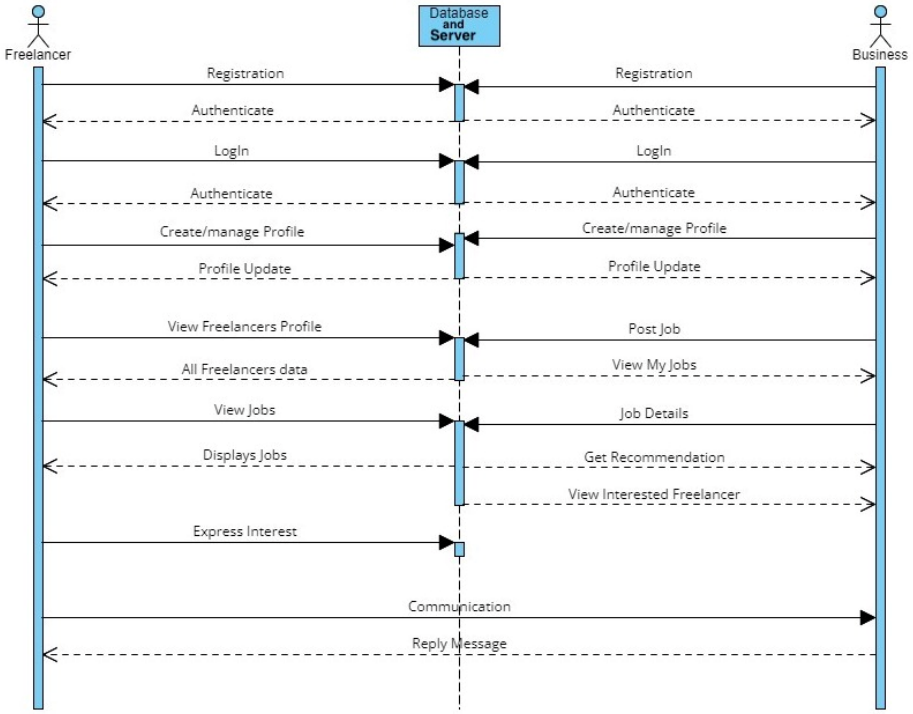


Figure [9]. *Sequence Diagram*

Chapter 5
Plan

Task	Duration	Task Description	Status
Preliminary Report	04/03	Submission of the preliminary report with detailed explanation of the project's aim and objective.	Completed
Development – Registration Process	07/03 – 14/03	Develop the front end and the back-end API Routes for Freelancer and business registration	Completed
Development – Profile Creation, add experience and education details, posting jobs	15/03 - 18/03	Develop the front-end and the back end of the application for freelancer and business profile creation, job posting,	Completed

		adding education and experience	
Development – Develop Dashboard view for freelancers and business.	19/03 - 31/03	Work on both the front end and back end for viewing jobs, post jobs and view other freelancers	Completed
Interim Report Submission	04/04 - 08/04	Interview with second marker	Completed
Research and development of recommendation system	08/04 - 15/04	Research and implementation of recommendation system	Completed
Develop Communication functionality between freelancers and business	08/04 - 15/04	Develop a feature for communication (Email communication)	Completed
Development – Add functionality for students	15/04 - 18/04	Add functionality specific to students. So, businesses can identify students as freelancers	Completed
Development – To implement recommended and optional features	25/04 - 06/05	Try to implement the recommended and optional features	Completed
Final Report and Viva Preparation	25/04 - 20/05	Working on the final report and viva presentation	Completed

Approach

In the initial phase of the project, the author was given the opportunity to select own topic of interest. The project idea the author produced was to develop a web application using a famous technology stack. The application idea was to develop a platform which can help freelancers to work on real time projects with ease and comfort. The additional idea was to bring students into the application, this helps students gain experience by working on real-time projects.

The Project started by achieving the essential requirements one by one. To do this, the author started designing the frontend and backend of the application simultaneously. To start with, the server was set up and the connection to the database has been achieved. The next step was to set up the API routes for registration, profile creation, dashboard implementation and so on. The proceeding step after that was to create models for jobs, creating API routes for posting jobs, viewing jobs and other job interaction API routes such as expressing interest if the freelancer likes the job and many more. After achieving all these the author went on to implement the front end of the application using react. The implementations will be further explained in detail in chapter 6 of this document.

After achieving more than half the requirements, the author went on to implement functionalities specific to students such as add their education details to their profiles, showing the recommended courses to improve their skill set.

Although there are changes which have been implemented into the application, the author was able to achieve the application's business logic and has successfully provided a business value to the work.

The testing of API routes has been achieved through postman the explanation on how postman helped the author to test the API routes has been detailed in chapter 6 of this document.

Methodology

There are several software development methodologies which can be used based on the project size and requirement. Some of the software development methodologies include. (Majewski, 2019)

- Waterfall
- Feature Driven Development
- Agile
- Scrum
- Extreme programming and
- Lean

The author, after understanding how each methodology works, found out that the scrum development approach is best suited for the development of applications with intermediate complexity. (Majewski, 2019)

What is Scrum Methodology?

Scrum is known as an alternative approach to agile methodology. In agile the progress is updated by making frequent client interactions. In Scrum, frequent interactions take place but within the team members. (Majewski, 2019)

To follow scrum approach, we must break down goals into sub goals and work towards each sub goal at a time. In Scrum approach the important part is to organize meetings, in our project scenario the scrum master is the author's project supervisor. The meetings are organized weekly, and the requirements are segregated into parts. When the meeting commences the author explains how he has achieved the requirements and receives feedback from the supervisor and makes changes based on the supervisor's feedback. (Majewski, 2019)

By following Scrum approach, the author was able to implement the features in time and has implemented the changes suggested in each phase of the software development. (Majewski, 2019)

Chapter 6

Implementation

The “Job recommender system for freelancers” was created by applying the knowledge obtained in the study of MERN stack. MERN stack involves a set of technologies which contributes to developing a whole web application.

The Requirements

The “Job recommender system for freelancers” consists of two different user types. One is the “Freelancer” who are either self-employed or university students who are eager to display their skill sets. The other is “Business” who are looking for skilled developers to work on their projects. Admin is responsible for monitoring the freelancers' profile, business profile and jobs within the application and can also delete them if necessary.

By Going through similar application, the author came to an understanding that the process of finding job is getting complicated for freelancers because of the bidding process. Our application will suggest freelancers to jobs based on the skill requirement. To achieve this the author has implemented a content-based recommendation system within the application.

Based on research and understanding the features which need to be implemented within the application, it has been listed below as user stories.

- Freelancers and business should be able to create an account by registering within the application.
- Freelancers and businesses should be able to create their profiles within the application after registration.
- Business should be able to post jobs by providing the necessary details such as the skill requirement.
- Business should be able to view the list of freelancers who are available.
- Business should be able to view the jobs posted by them
- Business should be able to get a list of recommended freelancers for the job.
- Business should be able to view the freelancers who expressed interest in their job.
- Business should be able to communicate with recommended freelancers and interested freelancers.
- Freelancer should be able to create profile by providing details such as portfolio, skill set, experience, education, and LinkedIn.
- Freelancer should be able to view the list of jobs posted by businesses.
- Freelancer should be able to express interest in the job if they prefer to work on the job.
- Freelancers should be able to edit their profile details whenever necessary.

Development

In this section the author will be explaining the development of functionalities within the application. The aim is to provide a precise explanation of how each requirement is achieved using

the MERN stack development approach. If any third-party library is used, that will be explained as well in detail.

The Project structure is divided into two sections, one is the “Client,” and the other is “Server”.

In the coming section the author will be explaining the following.

- Back-end Development
- Front-end Development

Back-end Implementation

The server folder of the application is structured into 4 categories, Routes (API), Middleware, config and server.js file. The *package.json* file contains all the dependencies that are required for the application to run.

MongoDB setup

The initial step was to create a Mongo DB cluster by using Mongo DB atlas. MongoDB atlas is a fully managed database service where the deployment process is automated. (MongoDB, n.d.)

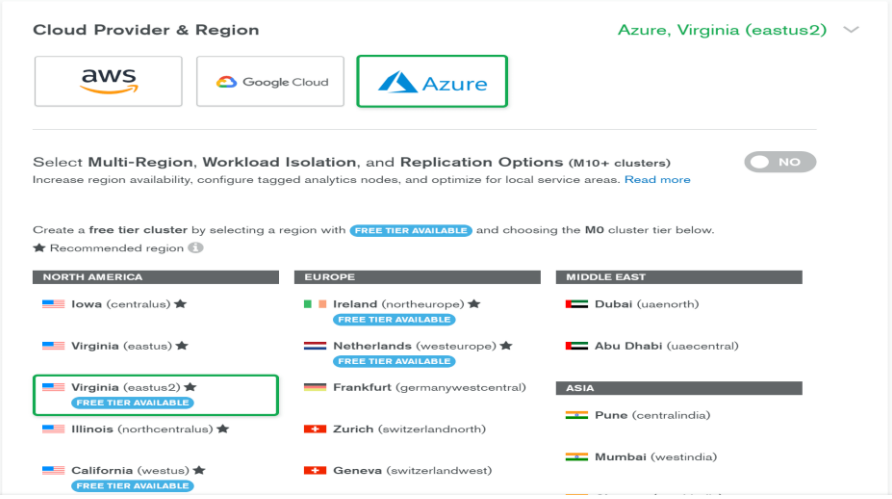


Figure [10]. *Creation of MongoDB Atlas cluster.*

MongoDB atlas automates the process of

- Infrastructure Provisioning
- Deployment and
- Setup

As shown in the figure, we must select the cloud provider, memory, region, and the size of instance. We can also perform additional configurations in the cluster using the API. (MongoDB, n.d.)

Atlas is an online database service, once the cluster is created, we can add users to the cluster and use the *MONGO_URI* to connect to the database. We can browse the database collections whenever required. (MongoDB, n.d.)

The *Mongo_URI* is added to the config file in the application and used for connecting to the database. Below code shows the function created to connect to the database. (MongoDB, n.d.)

When the connection is successful, the terminal displays "Database is Connected".

Server Setup

The following step is to set up the server using the Express framework. Below code "Server.js" is the entrance file to the application.

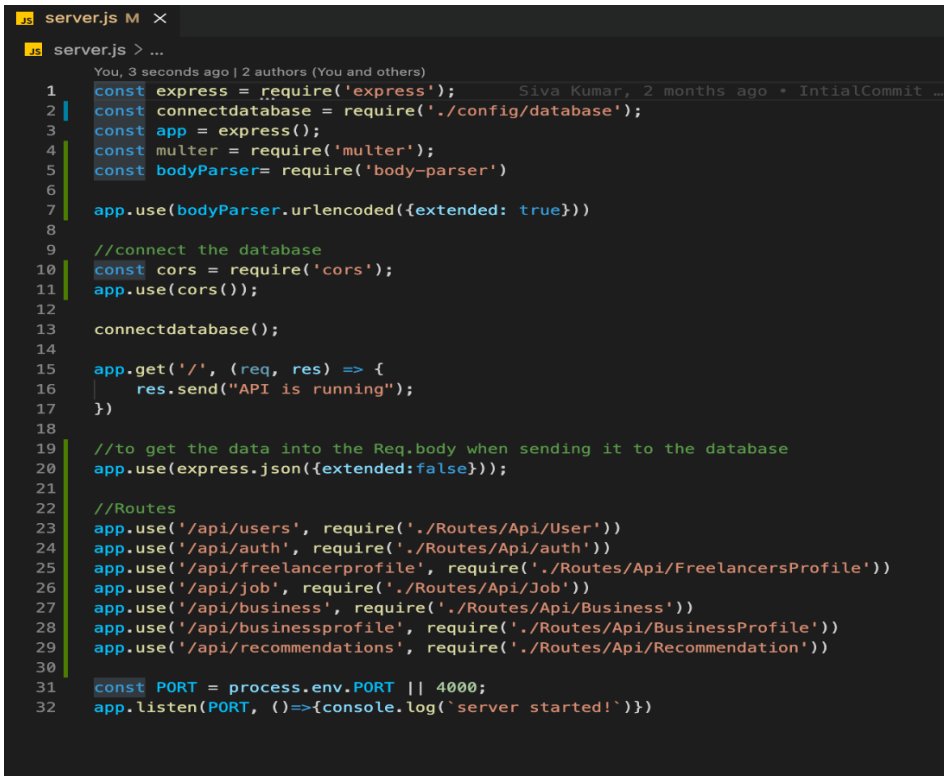


Figure [11]. *Server.js code*

The routes are defined using the "*app.use()"* which takes in two arguments. The actual route and the path to the file which contains the API routes. The server is setup to run on the PORT 4000 and the "*connectdatabase()*" function is called which connects the database to the server.

Mongoose Schemas

Once the server is set up and the database is connected, the following step was to create mongoose Schema.

There are 5 schemas used in the application.

- Freelancer
- Freelancer Profile
- Business
- Business Profile and
- Jobs

All the schemas are not shown in this document. Below is the Mongoose Schema of Jobs all the other schemas are created in the same way. Only the properties change based on the schema usage.

Figure [12]. *Job Schema*

Freelancer and Business Schema

The freelancer and the business schema consist of properties which are required from the users to register. The freelancer and business models only accept properties which are defined in the schema. Only these properties can be added to the database.

Password is not saved in the database directly, instead the author is saving the password in the database after encrypting it.

Freelancer and Business Profile Schema

The freelancer profile and business profile consist of properties which are required to create a profile within the application.

To associate this model to the actual user model (Freelancer and Business Schemas) we are using property type of "*mongoose.Schema.Types.ObjectID*" which connects to the id of actual user models.

Job Schema

As mentioned earlier, the job schema is associated to the business by using the property type "*mongoose.Schema.Types.ObjectID*" . Job Schema also consists of the list of interested freelancers for the job which is associated to the user (Freelancer Schema) model.

Encrypting the Password

```
//encrypting the password

const salt = await bcrypt.genSalt(10);

user.Password = await bcrypt.hash(Password,salt);

//saving the user to the database
await user.save()

const load ={
    user:{
        id : user.id
    }
}
```

Figure [13]. *Password encryption using bcrypt*

Once the user provides the password, it is hashed using the *bcryptjs* node module. *Bcryptjs* is used for storing hashed passwords instead of plain text. To hash the password, we use "*bcrypt.genSalt()*" and pass in the password as arguments to *bcrypt.hash()* function. Which encrypts the password, and the encrypted password is saved to the database.

Using JSON web token for Authentication

```
//JWT sign
    jwt.sign(load, config.get('jwtsecret'), {expiresIn: 560000},
    (err, token) => {
        if(err) throw (err);
    //sending the token which is received back.
        res.json({token});
    });
```

```
if(!token){
    return res.status(401).json({msg: "Token missing Authorization denied"});
}
try{
    const decoded = jwt.verify(token, config.get('jwtsecret'));

    req.user = decoded.user;
    next();
}
```

Figure [14]. *Using JWT sign () and Verify () for authentication*

Once the user is registered with the application, we need to return a JSON web token. These tokens can be used to authenticate the user and access protected routes in the system. To generate the JWT token it requires a payload, the payload can be anything related to the user, in our application the author is using the id of the user as a payload.

To install JWT we have run the following command "*npm install jsonwebtoken*." Once it has been installed, we can use *jwt.sign()* to generate the web token and *jwt.verify()* to verify the web token when the user is trying to access protected routes.

We bring in JWT into the application code like the way we use node packages. Figure [10] consists of API route used to perform user authentication. As you can see from the code, the user id is sent as a payload, and JWT secret is required which is stored in the config file of our application and an expiration time for the JWT. If the time expires the user might have to relog into the application. Once the token is generated, it will be sent as a response. Once the user has the token, we can send it as part of the header when making the API request, this enables the author to authenticate the user when accessing protected routes.

Middleware

As explained above the *jwt.sign()* generates token, now we must send the token back for authenticating the user to access protected routes. To achieve this the author has created custom middleware. Since we are using middleware, it taken in three arguments (req, res, next). Once the request is complete, we can use next () which is callback that helps us move on to the next piece of middleware.

Middleware > Authentication.js > ...

```
const jwt = require('jsonwebtoken');
const config = require('config');


function middleware(req, res, next){
    //passing token to header
    const token = req.header('AuthenticationToken');

    if(!token){
        return res.status(401).json({msg: "Token missing Authorization denied"});
    }
    try{
        const decoded = jwt.verify(token, config.get('jwtsecret'));

        req.user = decoded.user;
        next();
    }
    catch(err){
            res.status(401).json({msg: 'Not a Valid Token'});
    }
};

module.exports = middleware;
```

Figure [15]. *Middleware code for authenticating the user*

When accessing protected routes we must send the token to the header, we have a request object which has a header property. "*AuthenticationToken*" is used as a header key to send the token. The JWT token needs to be decoded, to decode the token we use *jwt.verify()* which takes in the token from the header and the JWT secret which the author has used. When creating routes, we can protect the routes by passing in the middleware as a second parameter. This helps in verifying the user when accessing protected routes.

How JWT and Middleware work

- Post user login details
- Create JSON web token with secret
- Return JSON web token to the browser
- Create middleware for verifying the token
- Using middleware send JWT to the authentication header.
- Using middleware verify the JWT token and send user information
- Sending response back to the client

To summarize, whenever a freelancer or business registers or signs into the application a JWT token is generated, which is verified using the middleware when the user tries to access protected routes in the application.

Routes and APIs

Application Programming Interface (API) helps in making two programs interact with each other. In Software development sharing of data between two systems is an essential aspect for development. Whenever the user performs any sort of interaction within the application and API request is sent to the application's server. When the server receives the request, it analyzes the

request and undertakes the requested actions and sends a response back to the client. The above explanation summarizes how the REST API works.

REST is known "Representational State Transfer", which is a set of predefined logics that programmers use for developing their APIs. In REST Communication happens over HTTP like servers and browsers.

In our application there are serval route files, and each file has multiple private and public routes.

Route Files:

- Auth.js
- Business.js
- Freelancer.js
- BusinessProfile.js
- FreelancerProfile.js
- Job.js

The below tables include, the HTTP method of each route, the route path, middleware which is used to authenticate the user before accessing that route, Description and if the HTTP method is POST then the user might have to provide details so the input fields are specified for this route.

Freelancer.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
POST	api/freelancers	None	This is a POST route which registers the freelancers with the application. The password gets encrypted before being stored in the database. JWT token is generated to authenticate the user when accessing protected routes.	UserName, FirstName, LastName, Email, Password, Linkedln, Location, Age, About

Business.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
POST	api/business	None	This is a POST route which registers the business with the application. The password gets encrypted before being stored in the database. JWT token is generated to authenticate the user when accessing protected routes.	CompanyName, ContactName, ContactEmail ,Password, Location, CompanyDescription

Auth.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
GET	api/auth/freelancer	Freelancer Authentication	This route authenticates freelancers and returns the freelancers details which were provided during registration. The password alone is not sent as it is confidential.	None
GET	api/auth/business	Business Authentication	This route authenticates business and returns the business details which were provided during registration. The password alone is not sent as it is confidential.	None
POST	api/auth/freelancer	None	This route logs freelancers into the application by comparing the password and email. It generates the JWT token which is used to authenticate the freelancer.	Email and Password
POST	api/auth/business	None	This route logs business into the application by comparing the password and email. It generates the JWT token which is used to authenticate the business.	Email and Password

FreelancerProfile.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
GET	api/freelancerprofile/me	Freelancer Authentication	This route gets all the profile information related to the freelancer.	None
POST	api/freelancerprofile	Freelancer Authentication	Once the user is registered, they will be prompt to create their profile. This POST route helps in creating profiles for the freelancers by getting the necessary details.	Qualification, website, status, github username, social

			This route will create profile for freelancer and add it to the database.	media links
GET	api/freelancerprofile	None	This route gets all the freelancers, and their profiles present in the database. Which can be used to display freelancers to business	None
GET	api/freelancerprofile/:user_id	None	This route gets all the details of individual freelancer by using the id of freelancer. The freelancer id is received as a param.	None
DELETE	api/freelancerprofile	Admin Authentication	This route deletes the freelancer profile and the freelancer details from the database	None
PUT	api/freelancerprofile/experience	Freelancer Authentication	Once the profile is created, the freelancer will be able to add experience to their profile. This route helps in updating the freelancer profile by adding experience	Title, company, location, from, to, and description
DELETE	api/freelancerprofile/experience/:exp_id	Freelancer Authentication	Once the freelancer adds experience, they will be able to delete the added experience if needed. This route takes the id of the freelancer experience and deletes it from the database.	None
PUT	api/freelancerprofile/education	Freelancer Authentication	Once the profile is created, if the freelancer is student, he/she will be able to add education to their profile instead of experience. This route helps in updating the freelancer profile by adding education details	Course of study, university, location, from, to, and description
DELETE	api/freelancerprofile/education/:edu_id	Freelancer Authentication	Once the freelancer (Student) adds education, they will be able to delete the added education if	None

			needed. This route takes the id of the freelancer education and deletes it from the database.	

BusinessProfile.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
GET	api/businessprofile/me	Business Authentication	This route gets all the profile information related to the business.	None
POST	api/businessprofile	Business Authentication	Once the business is registered, they will be prompted to create their profile. This POST route helps in creating profiles for the business by getting the necessary details. This route will create a profile for business and add it to the database.	Website, Status, Established, clients, company category and employee count
GET	api/businessprofile	None	This route gets all the businesses, and their profiles present in the database.	None
GET	api/businessprofile/:business_id	None	This route gets all the details of individual business by using the id of business. The business id is received as a param.	None
DELETE	api/businessprofile	Admin Authentication	This route deletes the business profile and the business details from the database	None

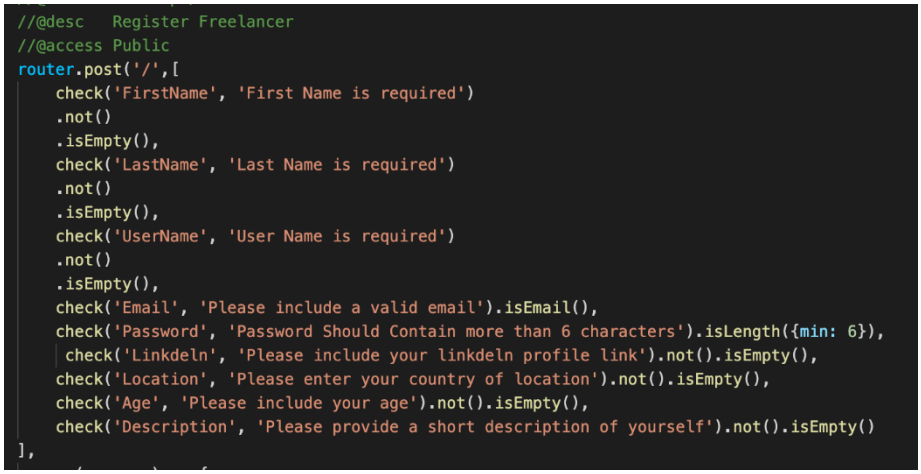
Job.js

Method	URL Endpoints	Middleware	Description	Input Fields if any
POST	api/job	Business Authentication	This route is protected, so business authentication middleware is used. This route adds job to the database by getting the required details. Once the job	Job title, job description, skill sets required,

			is posted it gets associated to the business profile.	job budget and job duration
GET	api/job	Business Authentication and Freelancer Authentication	This route is protected, so both the middleware in the system is used for authenticating the freelancer and business. This route gets all the jobs posted by business.	None
GET	api/job/:id	Business Authentication	This route gets the details associated to a single job. The job id is received as a param. This route is used to display all job-related details in a single page.	None
DELETE	api/job/:id	Business Authentication	This route helps the business to remove the jobs posted by them. Job id is received as a param.	None
PUT	Api/job/interested/:id	Freelancer Authentication	This route allows freelancers to express interest in a job if they prefer to work on the job. This add freelancers to the interested property of the job. Freelancer id is received as a param	None
PUT	Api/job/uninterested/:id	Freelancer Authentication	This route allows freelancers to remove expressed interest in a job if they prefer not to work on the job	None

To validate the input fields when requests like POST are sent to the server the author is using the express validator in the code. The validations are shown below in the screenshot.

```
const {check, validationResult} = require('express-validator');
```



```
//@desc   Register Freelancer
//@access Public
router.post('/',[
    check('FirstName', 'First Name is required')
    .not()
    .isEmpty(),
    check('LastName', 'Last Name is required')
    .not()
    .isEmpty(),
    check('UserName', 'User Name is required')
    .not()
    .isEmpty(),
    check('Email', 'Please include a valid email').isEmail(),
    check('Password', 'Password Should Contain more than 6 characters').isLength({min: 6}),
     check('Linkdeln', 'Please include your linkdeln profile link').not().isEmpty(),
    check('Location', 'Please enter your country of location').not().isEmpty(),
    check('Age', 'Please include your age').not().isEmpty(),
    check('Description', 'Please provide a short description of yourself').not().isEmpty()
],
```

Figure [16]. *Using Express Validator to validate user inputs.*

Postman

When starting with the backend of the application, there is no front end available to test these API routes created. The author uses third-party tool known as Postman to check the API routes

What is Postman?

Postman is an application which helps in checking the API Routes. It acts as an HTTP client to check the HTTP requests. Using postman, we can receive different/diverse types of responses. (Romero, 2021)

Postman endpoint interaction methods:

- GET
- POST
- PUT
- DELETE
- PATCH

Postman sends response codes in return when checking the API routes. We can receive several types of responses based on the API testing being performed. (Romero, 2021)

- Response status vary from status code 100 – 500, each representing a response for the request sent.

To help us organize tests, postman provides the opportunity to group requests together known as “collections”. Collections is a folder where we can store the request and organize the request as per our needs. Postman is very efficient when the development approach is agile. (Romero, 2021)

This section will contain more details about how author used postman to check the API routes created based on the user requirements. (Romero, 2021)

Every single API route mentioned above has been tested using the postman, below screenshot shows how the APIs have been tested.

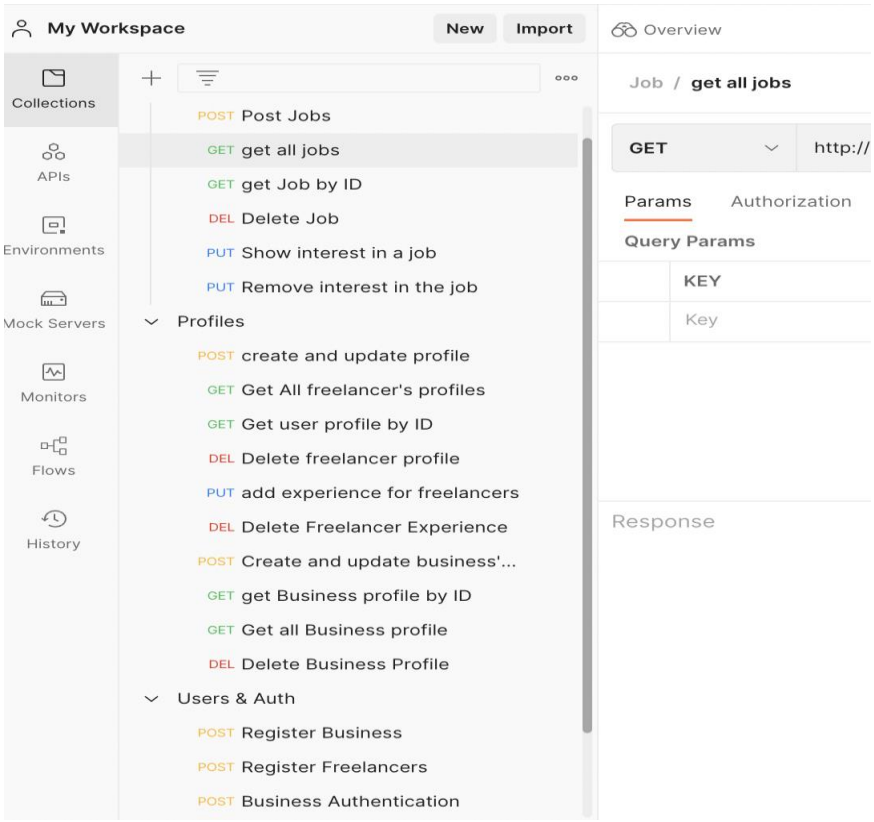


Figure [17]. *Saved API Routes for testing the end points.*

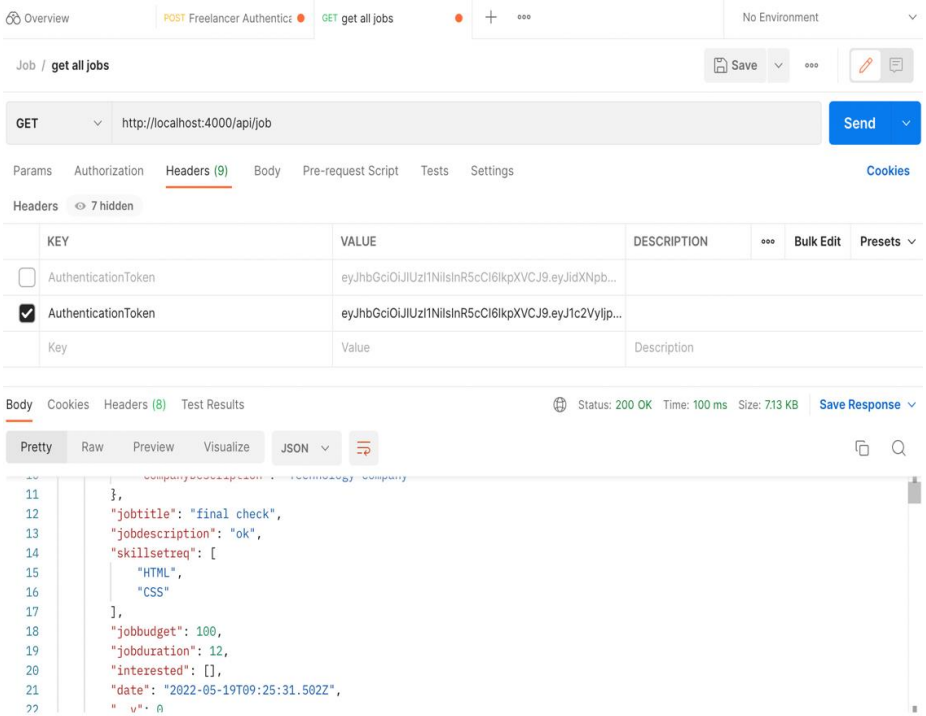


Figure [18]. *Testing the GET request for all jobs using Postman API*

Example:

As u can see in the above screen shot, the GET request is sent to get all the jobs posted by business. This is a private route, only freelancers and business when authenticated can view this route. So the token is added to the header, now we receive the response for the request as a JSON object containing all the properties present in the Job schema.

Front-end Implementation

Back-end implementation of the project involved the use of multiple technologies. The front-end of the application has been implemented using React.js. Due to the scope of the dissertation the author will not be explaining the front-end implementation in detail. The aim is to precisely discuss some components that are important in understanding the implementation of the project.

To start with the author used the command "*npm create-react-app devhub*" to create a react application.

The initial step is to install dependencies into the *package.json* file by running the command "*npm install packagename*".

Some of the dependencies in the project:

- "moment" - which formats the dates based on the developer's requirements.
- "React-redux" - Application state is managed by an open-source JavaScript library known as the React-Redux.
- "React-bootstrap" - The UI of the application is designed using react-bootstrap.

Routing

In the proceeding step, to setup router the author installed "*react-router-dom*" package from the *npm registry*. The react router is used in the app.js file present in client side of the application. The author has defined the <*Router*> component followed by <*Routes*> and under <*Routes*> we can define all the routes and the component that needs to be rendered when accessing that route.

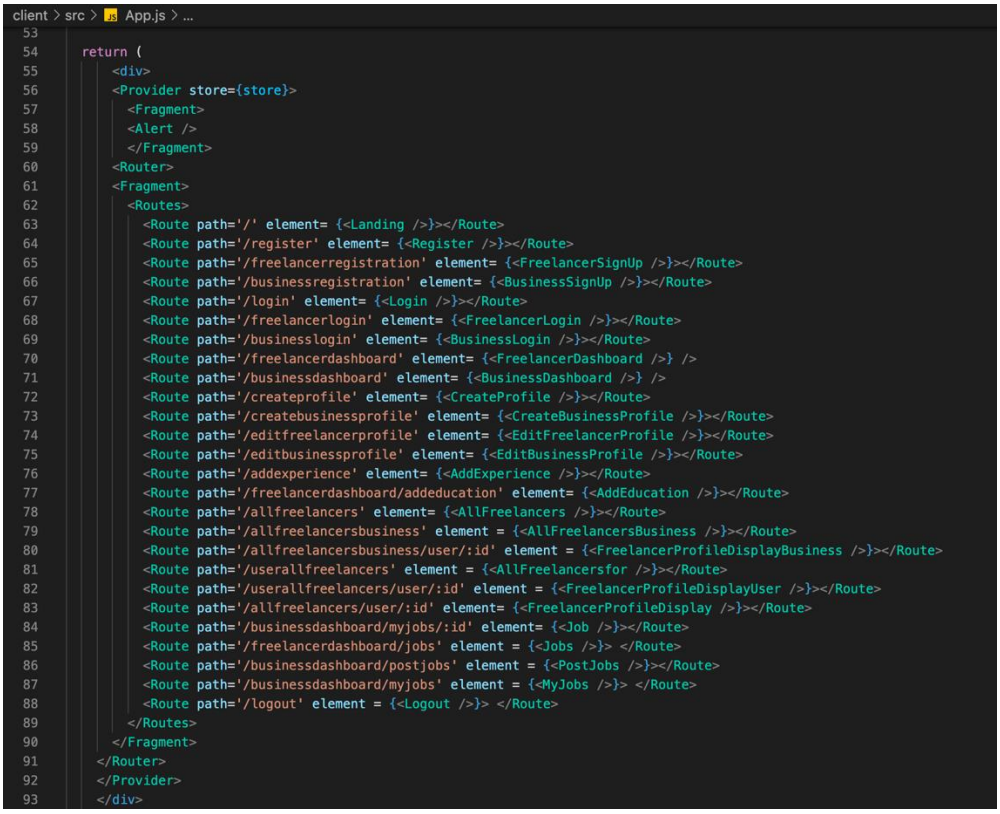


Figure [19]. *Routes defined in app.js*

For example, when you have look at the above screenshot for the path *'/freelancerregister'* the component to be rendered is mentioned as *element= {<FreelancerSignUp />}*. When the user tries to access the route, the Freelancer Sign Up component will be rendered.

Private Route

A Component is created to check whether the user is authenticated or not. The authentication state of the user is passed on to the components using react redux. The routes which are to be protected are wrapped into the private route, so only authenticated users can access the routes such as dashboard etc.

React Redux

As explained in the earlier chapters, react redux is used in the client side of the application. Which helps in passing and managing the applications state between the components.

There are other similar state managers used in react application such as the context API, for this project the author is using quite a few reducers so react redux would be an ideal choice.

Component-Level State

```
const [formdetails, setFormDetails] = useState({
  Email:'',
  Password: ''
});
  const {Email, Password} = formdetails;

  const onChange = e => setFormDetails({...formdetails, [e.target.name]: e.target.value})
```

Figure [20]. *Using useState hook in Feelancerlogin component*

The data present in the login and register components is maintained as a component level state. So, by using the react hook known as *useState ()* component level state is maintained.

Application-Level State

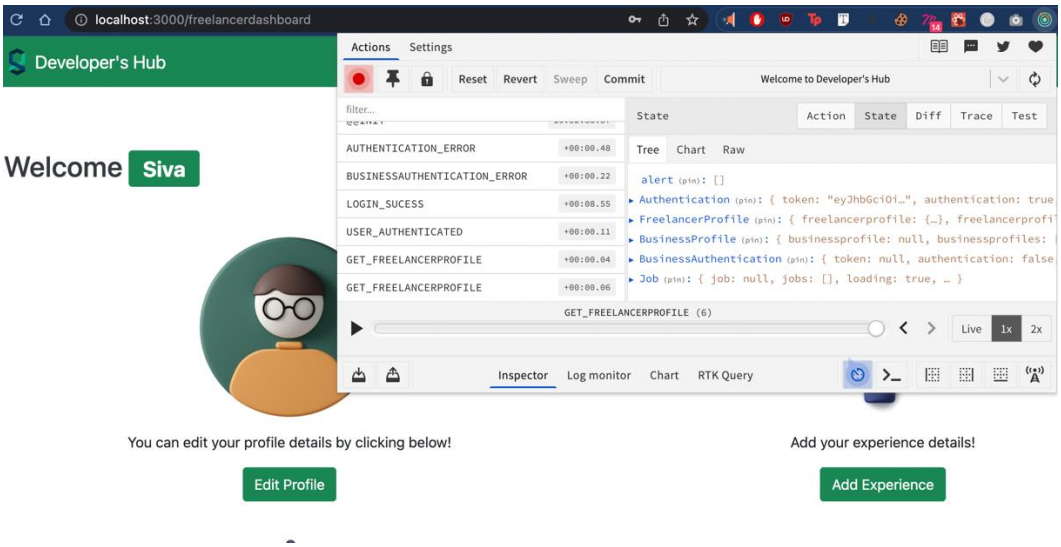


Figure [21]. *React Redux Store*

As explained in the above paragraph, the registration and signup states can be maintained as a component level state but when it comes to freelancers' profile and jobs, the state should be maintained at the application level as it should be accessible to several other components that might use them to render content to the UI. This is where redux is useful, it acts as a cloud that floats over the application to which we can submit actions through events and make the data flow down to any component.

How Redux Works in our application?

Store

For example, the freelancer profile data from the server is stored in something called the "*Redux Store*", this is achieved by calling an action that can fetch the data with the help of axios and saves the received data to the store.

client > src > JS Store.js > ...

sivakumar4698, 3 weeks ago | 1 author (sivakumar4698)

```
1   import {createStore, applyMiddleware} from 'redux';
2   import {composeWithDevTools} from 'redux-devtools-extension';
3   import thunk from 'redux-thunk';
4   import rootReducer from './reducers';
5
6   const intialState = {};
7
8   const middleware = [thunk];
9
10  const store = createStore(
11      rootReducer,
12      intialState,
13      composeWithDevTools(applyMiddleware(...middleware))
14  );
15
16  export default store;
```

Figure [22]. *Store.js code*

Reducer

Reducer is a function that takes in an action, we can dispatch an action to the reducer. The reducer decided on how to handle the state and passes it down to the components in the UI. It also updates any other component that uses that piece of state.

From any component we can call "actions" to perform something. For example, let's say the freelancer wants to update profile then the following steps take place.

Initially, we must pass down a request to the server and update the database. This is achieved by calling an action which uses axios to hit the API endpoints and gets the response.

Once the above step is done, we need to update the user interface by updating the state in store, this is achieved through reducers**.**

Actions

Actions are functions that update the reducer with the response received from the axios requests. The action functions are passed down as prop types to components which need them, when these functions are called in the components the reducer gets updated with the response data which will be reflected in the store.

Redux Dev Tools

Redux dev tool is a browser extension which helped the author to manage and debug state changes in the application. It shows the actions that are being fired when the user performs some interactions in the UI. It Shows the complete data that is available to use. The redux Dev Tool is open sourced which means it is easy to add it as a browser extension and use it.

The application state and data are maintained using the react redux. The recommendation system built in the application is built using the data received from the redux store. To summarize,

- When user performs an interaction in the application – action gets called
- The action gets dispatched to a reducer
- Reducer decides on what to do with the state
- State is passed down to the components of the application

Actions Created

How are actions created?

All the actions in the application perform an axios request to hit the API endpoints. Depending on the HTTP method such as GET, POST, PUT, and DELETE the axios request changes. For example, when the user tries to sign up with the application *axios.post()* request is sent to the API endpoint and the required input is sent as parameter.

What is Axios?

Axios is part of the JavaScript library, the author is using axios to make HTTP requests to the API endpoints created using the *Node.js*. All the CRUD operations in HTTP request can be performed using axios. Another alternative for axios in *react.js* is *fetch (),* fetch comes with limitations where the request and response are mixed making it complicated for use.

For example, See the below screenshot on how the axios request is performed. when the user signs up using the client side of the application.

```
//register freelancer
export const freelancersignup = ({UserName, FirstName, LastName,Email, Password,
    confirmpassword, Linkdeln, Location, Age, Description}) => async dispatch => {
        const Configuration = {
            headers:{
              'Content-Type':'application/json'
            }
          }

          const Content_Body = JSON.stringify({UserName, FirstName, LastName, Email, Password,
            confirmpassword, Linkdeln, Location, Age, Description});

        try{
          const response = await axios.post('http://localhost:4000/api/users', Content_Body, Configuration);

          dispatch({
              type:SIGNUP_SUCESS,
              payload: response.data
          });

          dispatch(loadFreelancer());

        }
       catch(err){
          const errors = err.response.data.errors;

          if(errors){
              errors.forEach(error => dispatch(DoAlert(error.msg, 'primary')));
          }
        dispatch({
            type:SIGNUP_FAIL
       });
       }

    }
```

Figure [23]. *Freelancersignup () action*

A function *freelancersignup()* is created and the required input fields are passed as a parameter. When making axios request like POST, "*content-type*" is sent as part of the header and the "*content-body*" contains all the fields which are required from the user. Once the response is received the

action *SIGNUP_SUCCESS* is dispatched to the reducer and the reducers passes the state down to the components.

```
switch(type){
    case SIGNUP_SUCESS:
    case LOGIN_SUCESS:
        //storing the token received into a local storage
        localStorage.setItem('token', payload.token);
        return{
            ...state,
            ...payload,
            authentication: true,
            loading: false
        }
```

Figure [24]. *Actions in Reducer*

Action files created are:

1.Alert.js

The action *DoAlert()* is created which is a function that can be called when an alert must be rendered in the front end of the application.

```
JS alert.js
client > src > actions > JS alert.js > ...
1   import {DO_ALERT, UNDO_ALERT} from '../actions/actiontype'
2   import {v4 as uuidv4} from 'uuid';
3
4   export const DoAlert = (msg, alerttype) => dispatch => {
5       const id = uuidv4();
6       dispatch({
7           type: DO_ALERT,
8           payload: {msg, alerttype, id}
9       });
10
11      setTimeout(()=> dispatch({
12          type: UNDO_ALERT, payload: id
13      }), 4000);
14  }
```

Figure [25]. *Action created to dispatch alerts in application*

2.FreelancerAuthentication.js

Actions created in this file are called when the freelancer tries to sign up or login to the application.

```
//load freelancer
export const loadFreelancer = () => async dispatch => {
   if (localStorage.token) {
       AuthToken(localStorage.token);
   }

   try{
   const response = await axios.get('http://localhost:4000/api/auth/freelancer');

   dispatch({
       type: USER_AUTHENTICATED,
       payload: response.data
   });
   }
   catch(err){
      dispatch({
          type: AUTHENTICATION_ERROR
      });
   }
  }
```

Figure [26]. *Loadfreelancer action*

Actions in this file are:

- *LoadFreelancer()* - This function checks whether the freelancer is logged into the application before he/she tries to access the private routes. This is done by making an axios request to the URL to perform the GET operation. If the freelancer is authenticated, the token gets stored in the localStorage.
- *Freelancersignup()* - This event/function is explained above. This action is called when the user tries to sign up with the application.
- *LoginFreelancer()* - This action is called when the user wants to login to the application using the login component, this action gets called by sending the user email and password as parameters. Axios sends the request to the API endpoint and the response is sent back which is updated using the reducer and passed down to components.
- *Loggingout()* - This action is called when the user logs out from the application. Action type EMPTY_PROFILE is dispatched to the reducer. The reducer updates the state by removing the freelancer's details.

3.BusinessAuthentication.js

As explained in Authentication.js, the business authentication file also includes the same set of actions, but the actions are related to business.

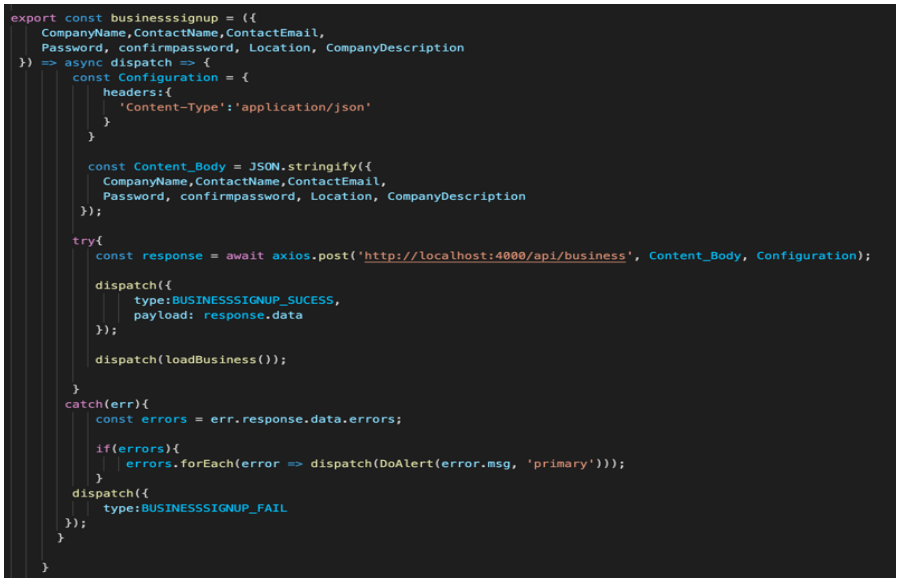


Figure [27].*Businesssignup action*

Actions in this file are:

All the actions in this file perform similar actions as mentioned in *FreelancerAuthentication.js*

- *LoadBusiness()*
- *Businesssignup()*
- *LoginBusiness()*
- *Loggingout()*

4.FreelancerProfile.js

Actions created in this file are used to hit the API endpoints related to the freelancer profile.

Figure [28].*getfreelancerproile action*

Actions in this file are:

- *getcurrentfreelancerprofile()* - This action gets called in the freelancer's dashboard which brings in all the details related to the logged in freelancer. All the received details from redux state are then rendered into the UI.
- *createfreelancerprofile()* - This action gets called in the freelancer's dashboard when the freelancer initially logs into the application. When they click create profile, this action gets called and the POST request is sent once the freelancer provides all the necessary information.
- *addexperience()* and *addeducation()* - These actions get called when the freelancer adds information to the dashboard such as experience and education.
- *removeeducation()* and *removexperience()* - These actions get called when the freelancer removes information from the dashboard such as experience and education.
- *Getallprofiles()* - This action brings in all the freelancer profiles, this action gets called when the business wants to view the available freelancers.

5.BusinessProfile.js

Actions created in this file hit the API endpoints related to the Business profile.

```
//get current freelancer profile
export const getbusinessprofile = () => async dispatch => {

    try{
    const response = await axios.get('http://localhost:4000/api/businessprofile/me');
    const responsess = await axios.get('http://localhost:4000/api/job');

    dispatch({
        type: GET_JOBS,
        payload: responsess.data
    });

    dispatch({
        type: GET_BUSINESSPROFILE,
        payload: response.data
    });
    }
    catch(err){
       dispatch({
           type: BUSINESSPROFILE_ERROR,
           payload: {msg: err.response.statusText, status: err.response.status}
       });
    }
   }
```

Figure [29].*businessprofile action*

Actions in this file are:

- *createbusinessprofile()* - This action gets called in the business dashboard when the freelancer initially logs into the application. When they click create profile, this action gets called and the POST request is sent once the business provides all the necessary information.

6.Job.js

Actions created in this file hit the API endpoints related to Jobs.

```
//get Jobs

export const getJobs = () => async dispatch =>{
    try{
        const response = await axios.get('http://localhost:4000/api/job');


        dispatch({
            type: GET_JOBS,
            payload: response.data
        });
        }
        catch(err){
            const errors = await err.response.data.errors;
            if(errors){
             Array.from(errors).forEach(error => dispatch(DoAlert(error.msg, 'primary')));
             }
           dispatch({
              type: JOBS_ERROR,
              payload: {msg: err.response.statusText, status: err.response.status}
           });
        }
       }
```

Figure [30].*getjobs action*

- *AddJob()* - This action gets called when the business post jobs, it hits the API endpoint responsible for posting jobs. Once the actions are called the response is sent to the reducer which updates or passes down the state to components.
- *AddInterested()* - This action gets called freelancers viewing job click interested button provided for each job. This adds the freelancer to the interested property of the Job, which is used further to display all the freelancers interested in the job.
- *getJob()* - This action is called when the business wants to view job details. When this action is called it hits the API endpoint and gets the details related to that job.
- *getJobs()* - This action is called to get all the jobs present in the database and display it to the freelancers where they can view jobs and express interest in the job.

Reducers in the Application

As discussed earlier, the actions that are defined are dispatched to the reducer and the reducer updates and passes the state down to the components.

Reducers maintain the state of the application, in each reducer file an initial state object is created. When the actions are dispatched to the reducer, the state object is updated based on the actions. The state object requires payload which is received from the response of axios call.

Reducer files:

1.Alert.js

Alert.js consists of two actions that update the initial state, one is “DO_ALERT” which performs alerts in the UI. Another one is “UNDO_ALERT” which removes the alert after “DO_ALERT” is performed.

2.BusinessAuthentication.js and Authentication.js

This reducer file updates the authentication state of both the business and freelancers respectively.

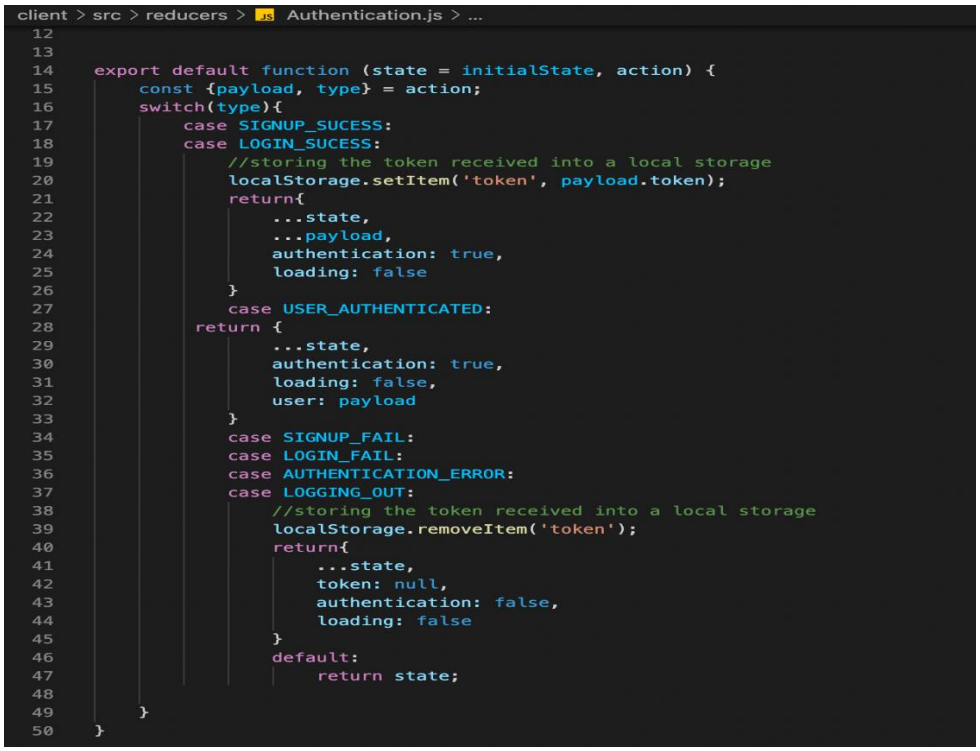


Figure [31]. *Reducers created for Authentication*

To show an example of how a reducer file works, see the above screenshot. The initial state object consists of details such as token, authentication, user, loading and they have been assigned initial values.

The actions that are dispatched to the reducer are given in switch case, based on the action the reducer updates the initial state. Whenever the author wants to view the application state, it is made available in the browser with the use of react redux dev tools.

Figure [32]. *USER_AUTHENTICATED action*

For example, if the action dispatched to the reducer is “USER_AUTHENTICATED” then the state gets updated as shown in the return statement of the switch case. Payload is sent to the user object. So, the JSON response received when performing an axios call is the payload and it is assigned to user. So, in the redux state all the information received from the response from server is stored in the state and is accessible.

3.BusinessProfile.js

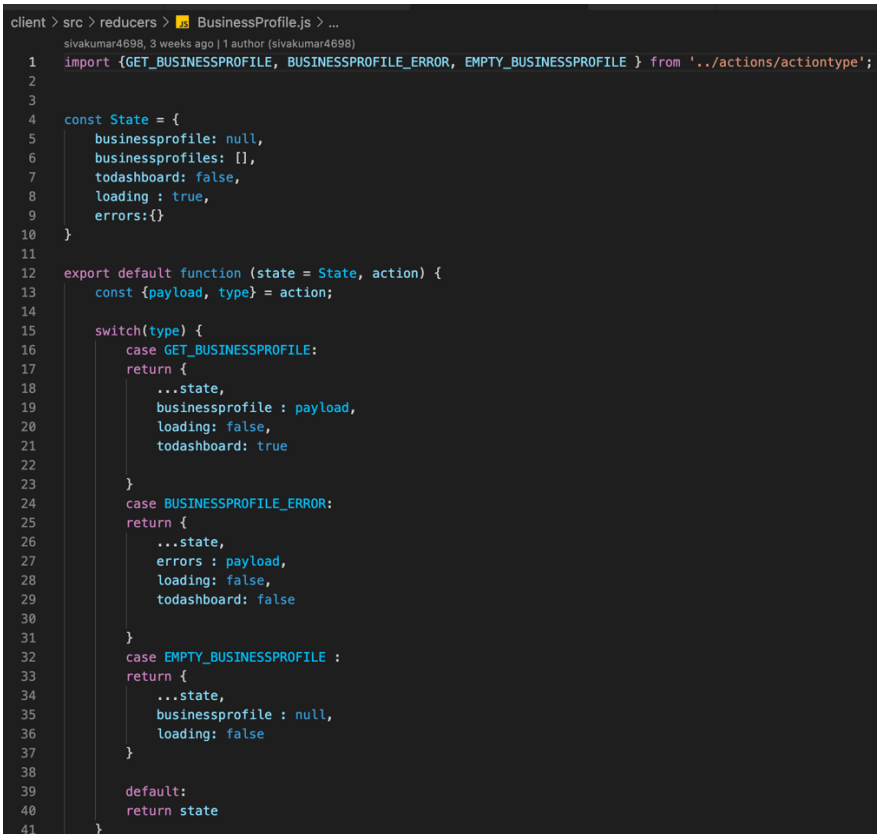


Figure [33]. *Freelancer Profile Reducers*

This reducer file receives actions related to business profile and updates the application state based on the response received.

4.FreelancerProfile.js

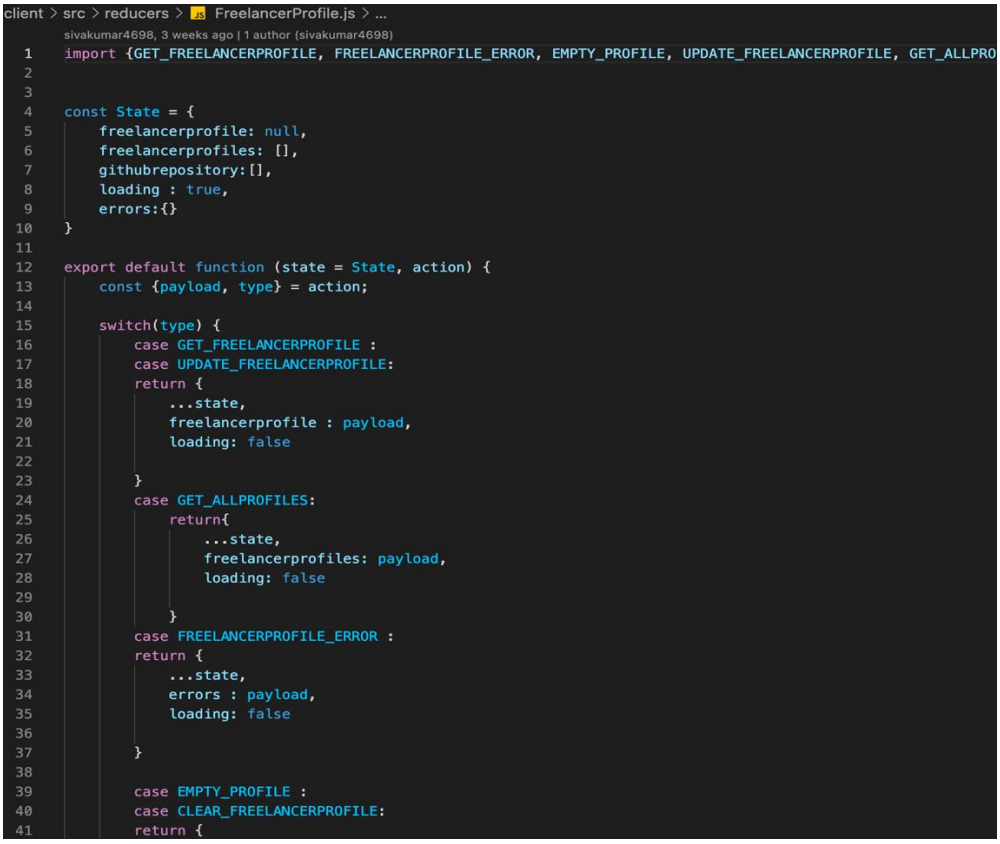


Figure [34]. *Business Profile Reducers*

This reducer file receives actions related to freelancer profile and updates the application state based on the response received.

5.Job.js

Like the above-mentioned schemas, this reducer file receives actions related to jobs and updates the application state based on the response received.

To summarize all the actions, reducers and store defined are part of the react redux state management. Where the response is sent to the server using action calls, the received response is dispatched to the reducer and the reducer updates the state. The state is made available between components based on requirement and the content is rendered to the UI.

React Components

A react application consists of various components, each component is a JavaScript function that returns elements that describe the UI. The components are segregated into folders. Each folder represents components that are related to registration, login, freelancers' dashboard, business dashboard, job displays and email communication. The code is maintained efficiently as the author has developed several reusable components

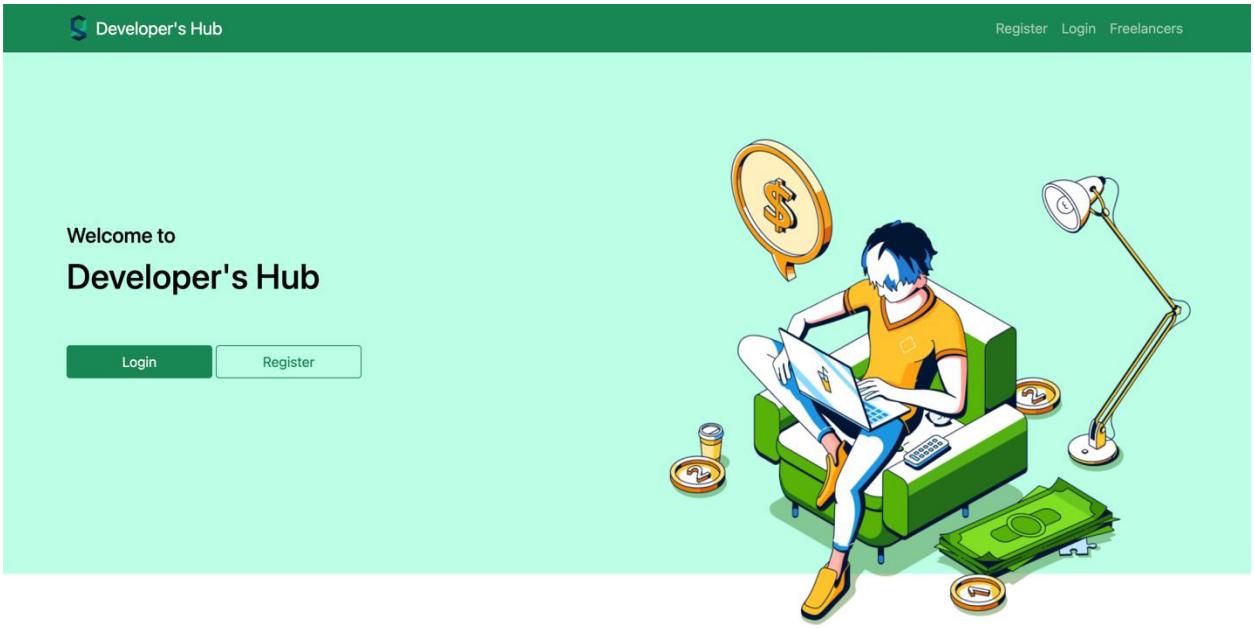
Styling

The author does not use CSS file for styling the components, instead react-bootstrap components are used for designing the interface. Any style specific to bootstrap components are defined in the

component tag itself using *style={{}}*. React icons are used to bring in icons as components into the UI components.

Landing Page

Landing.js component defines the user interface of the application's landing page. The landing page components consist of other components such as *<NavBar />* which defines the navigation bar interface of the application.



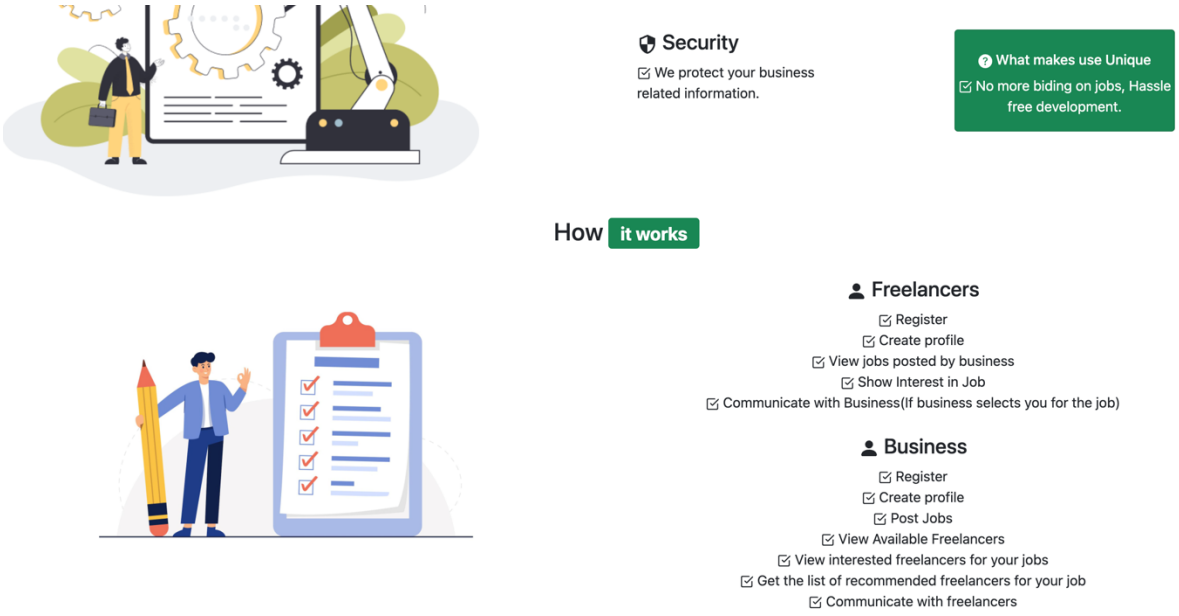


Figure [35]. *Landing Page*

Login Page

This Login.js component defines the user interface of the application's login page. The author uses *useState()* which is a react hook that updates the state based on the user input. *OnSbumit ()* the author calls *loginfreelancer()* action which performs the HTTP request to authenticate the user. This action is brought into the component by importing the action from the authentication.js action file and by using it as a *proptype*. Once the user authenticates the state gets updated in redux.

The login page appears based on the user types:

- Freelancer
- Business

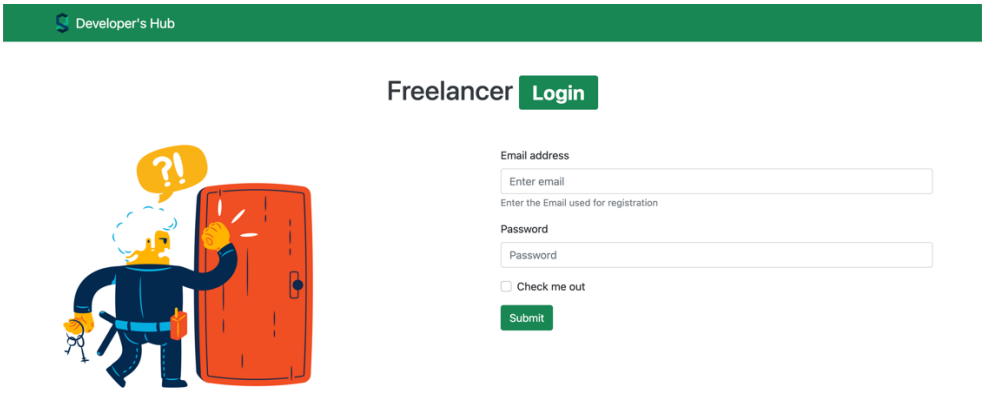


Figure [36]. *Login Page*

Signup page

This signup.js component defines the user interface of the application's sign-up page. The author uses *useState ()* which is a react hook that updates the state based on the user input. *OnSbumit ()* the author calls *businesssignup()* action which performs the HTTP request to register the user. This action is brought into the component by importing the action from the *businessauthentication.js* action file and by using it as a *proptype*. Once the user fills in the necessary fields and clicks submit, the user gets authenticated into the application and will be prompted to create a profile.

The signup page appears based on the user types:

- Freelancer
- Business

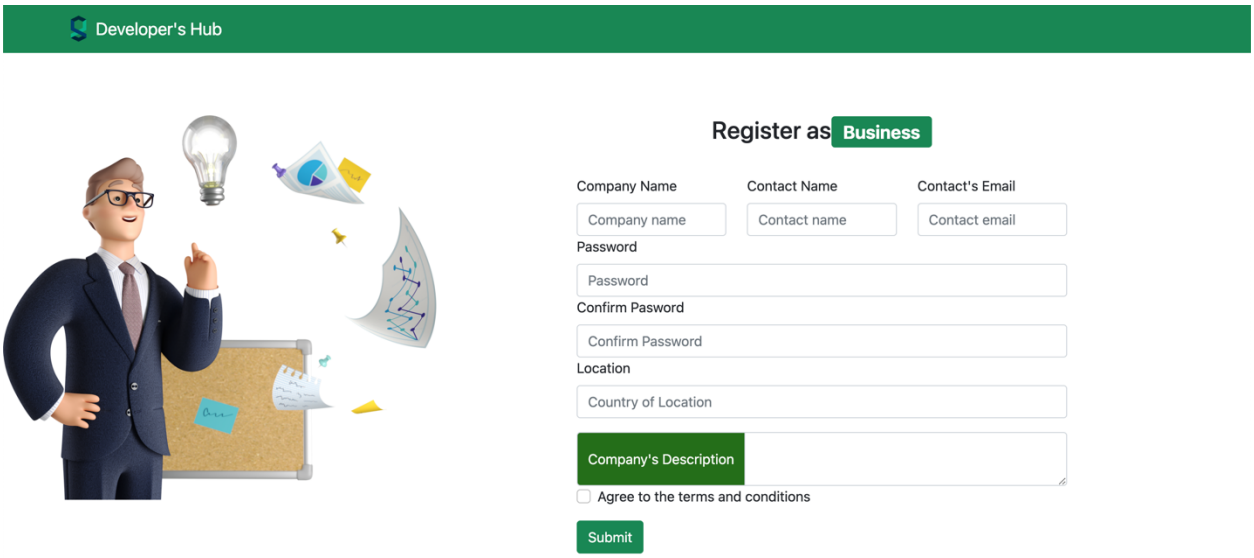


Figure [37]. *signup Page*

Create Profile Page

This createprofile.js component defines the user interface of the application's profile creation page. The input fields on this page change based on the user type. The author uses *useState()* which is a react hook that updates the state based on the user input. *OnSbumit ()* the author calls *createfreelancerprofile()* action which performs the HTTP request to create profile for the user. This action is brought into the component by importing the action from the *freelancerprofile.js* action file and by using it as a *proptype*. Once the user fills in the necessary fields they will be navigated back to the dashboard.

The create profile page appears based on the user types:

- Freelancer
- Business

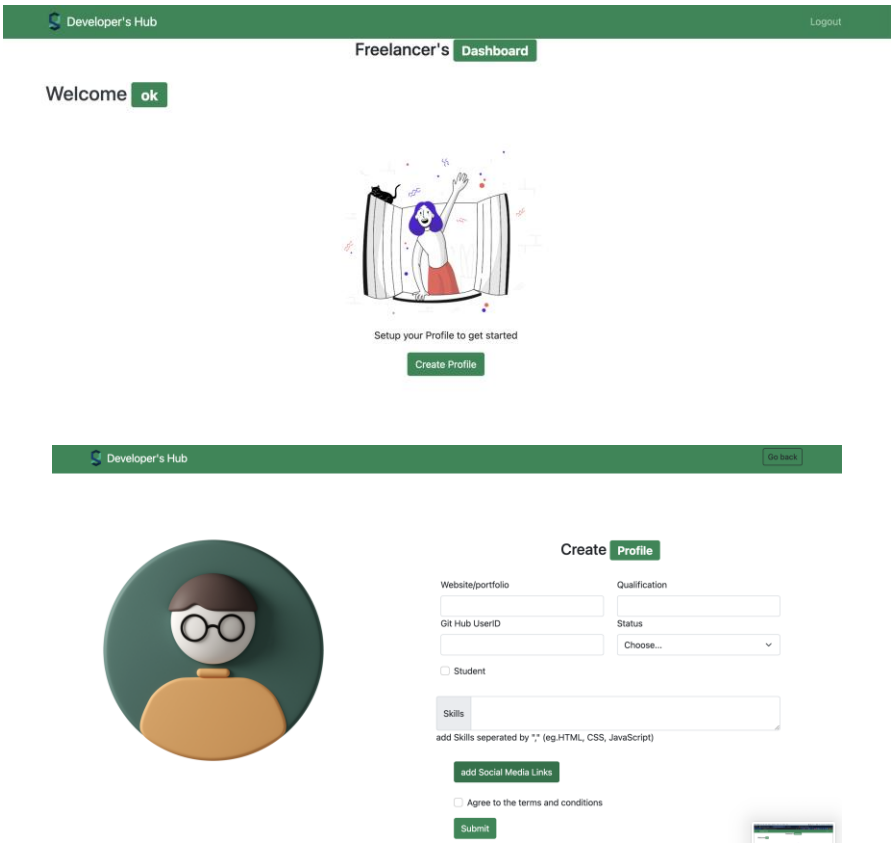


Figure [38]. *Create Profile Page*

Freelancer's Dashboard

Freelancerdashboard.js component defines the user interface of the freelancer's dashboard. This component involves functionalities such as

- Edit Profile
- Add Education/Experience
- View Jobs
- View other freelancers

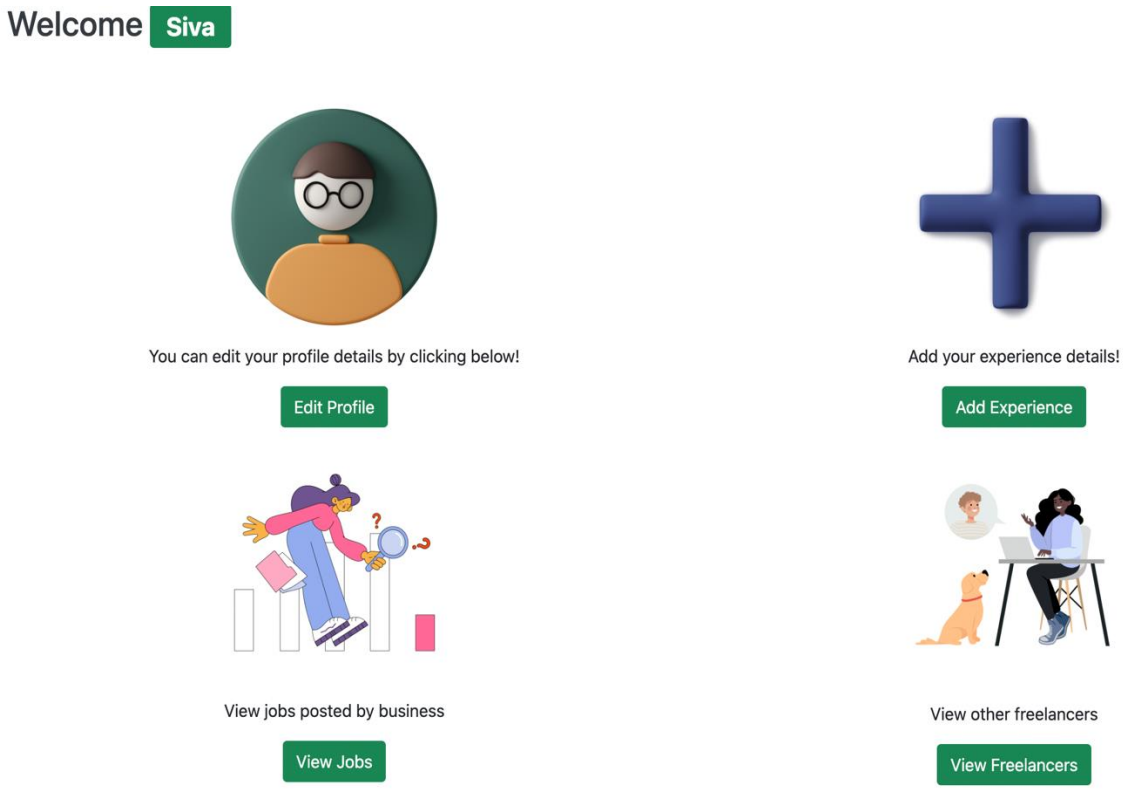


Figure [39]. *Freelancer's Dashboard*

Once the profile creation is complete the respective action which is called will update the state in redux. The state consists of details such as the freelancer profile and freelancer registration details. The navigation in the application takes place using react hook known as *useNavigate()* which is imported into components that have functionalities for navigating to different pages.

Edit Profile page

The user gets routed to *'/editfreelanceprofile*' which is defined as a route in app.js file. This route renders the component <*EditProfile* /> which contains the previously entered user inputs so that the user can edit the profile details whenever necessary. These changes get updated in the database as the author uses *useState()* and *useEffect()* react hooks. *UseState()* updates the changes in inputs, *useEffect()* calls the action which updates the profile details in the database. These details get updated in the redux state under the freelancer profile with the help of the reducer.

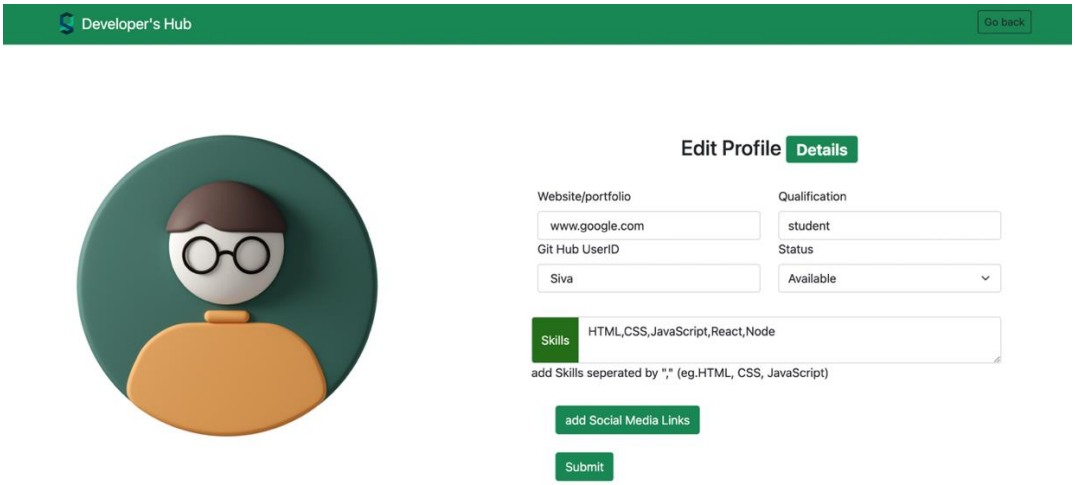


Figure [40]. *Edit Profile*

Add Education/Experience

If the freelancer is student, he/she can add education details. If the freelancer is self-employed, he/she can add their previous or current experience details.

Once the freelancer clicks Add Education or Add Experience, the user gets routed to *'/addexperience'* or '*/addeducation'* which is defined as a route in app.js file. This route renders the components *<AddExperience />* or *<AddEducation />* which contains fields where the freelancer can provide the necessary details. Once the freelancer submits the form, the *useState()* hook updates the *freelancerinputs* and the inputs are sent as a parameter to the action type which is called using *useEffect()* hook. This action performs the necessary HTTP requests and adds the freelancer's education or experience details into the database. This gets updated in the redux state which is later used to display the experiences or education details of the freelancer in the dashboard.

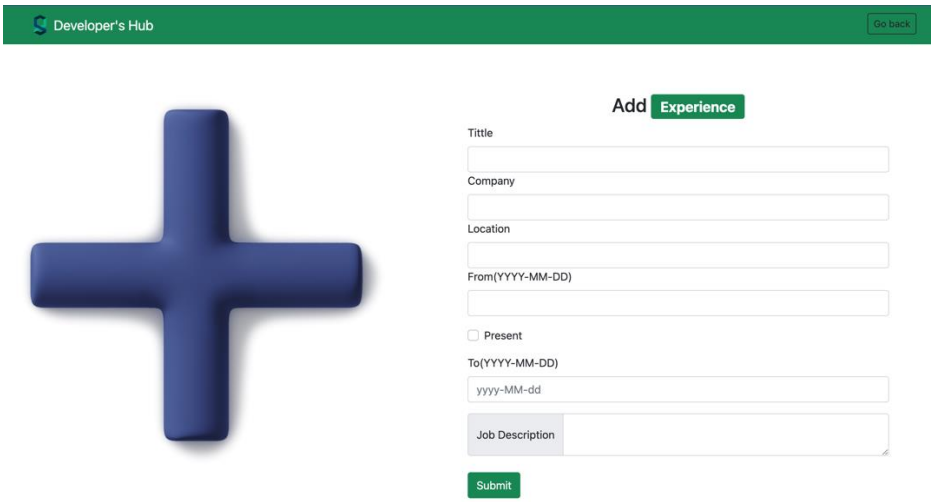


Figure [41]. *Add experience/Education*

View Jobs

The user gets routed to '*/freelancerdashboard/jobs*' which is defined as a route in app.js file. This route renders the component *<Jobs />* which renders another component called *<JobView />.* The <Job /> component consists of all jobs whereas the *<jobView />* defines the interface of each job.

Once the freelancer clicks view jobs, the <Jobs /> component imports *getjobs()* action which sends the HTTP request to the server to get the list of all jobs. The *getjob()* action is imported as *proptype* and called using the react *useEffect()* hook. The jobs posted by business appear in the redux state, which is then passed into the *<jobView />* component which creates an interface for each job and displays it.

Freelancers also can express interest if they prefer to work on the job. This is achieved by creating a button that *onClick()* calls an action *addinterested()* that sends the HTTP request to add the freelancer to the job's interested list. This gets updated in the database and in the redux state. Freelancer will also be able to view business related details.

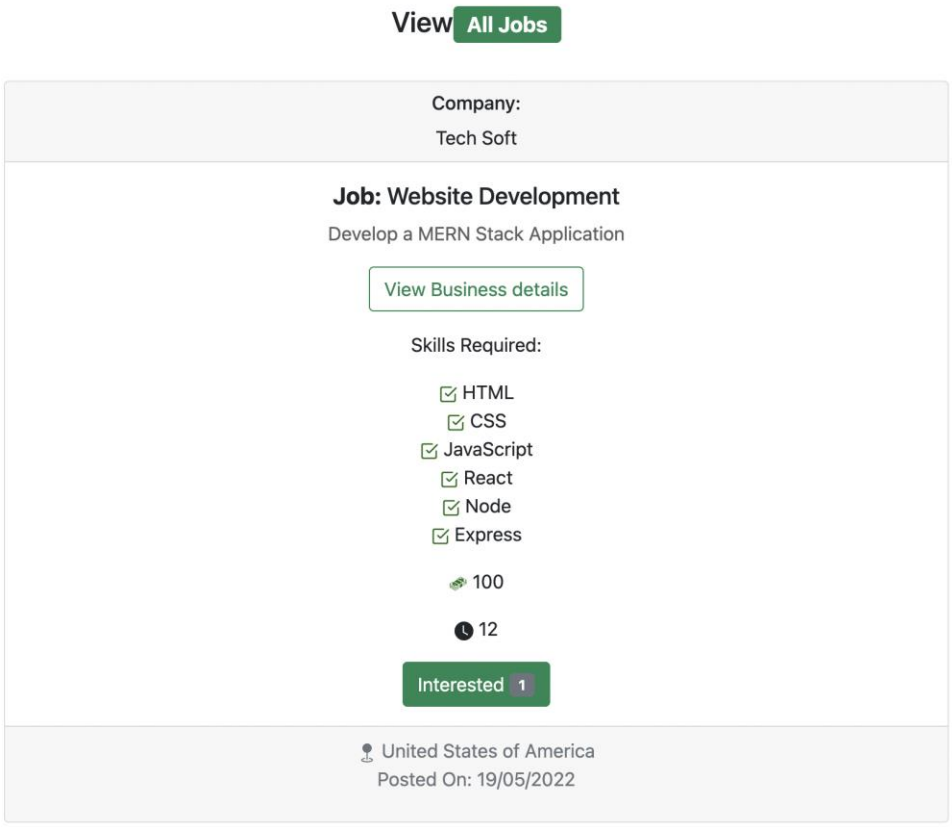


Figure [42]. *Add experience/Education*

View other freelancers

The user gets routed to '*/freelancerdashboard/allfreelancers*' which is defined as a route in app.js file. This route renders the component *<AllFreelancers />* which renders another component called *<FreelancerProfileItem />*. The < *AllFreelancers />* component consists of all freelancers whereas the *<FreelancerprofileItem />* defines the interface of each freelancer profile.

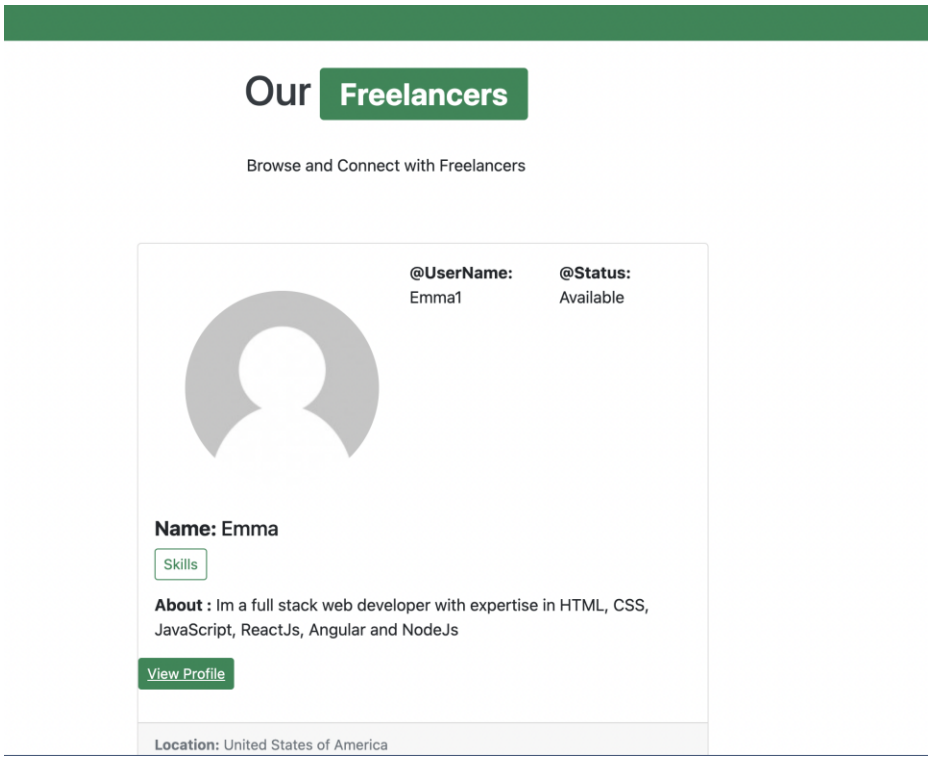
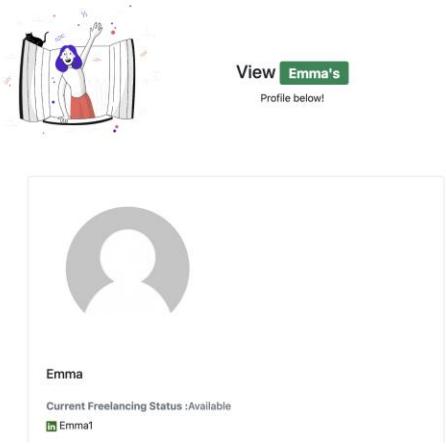


Figure [43]. *View all freelancers*

Once the freelancer clicks view other freelancers, the < *AllFreelancers* /> component sends *getallprofiles()* action which is called using the *useEffect()* hook. which sends the HTTP request to the server to get the list of all freelancers from the database. All the freelancers appear in the redux state, which is then passed into the < *FreelancerprofileItem* /> component which creates an interface for each freelancer and displays it.



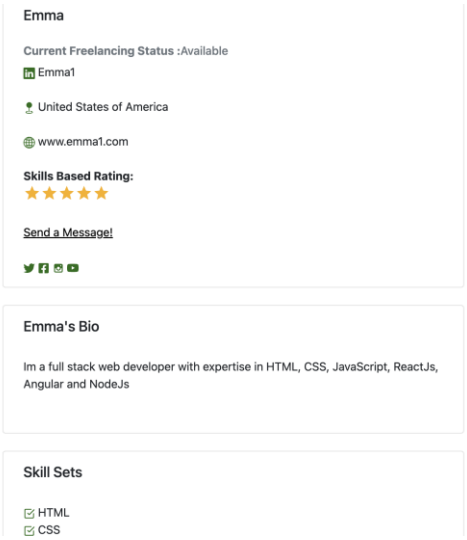


Figure [44]. *View freelancer details*

The freelancers appear as cards, when the user clicks view details, it renders another component *<FreelancerProfileDisplay />* which contains detailed information of freelancer such as experiences, GitHub profile link, freelancer rating based on skills.

Business Dashboard

BusinessDashboard.js component defines the user interface of the Business dashboard. This component involves functionalities such as

- Edit Profile
- View Freelancers
- Post Jobs
- My Jobs

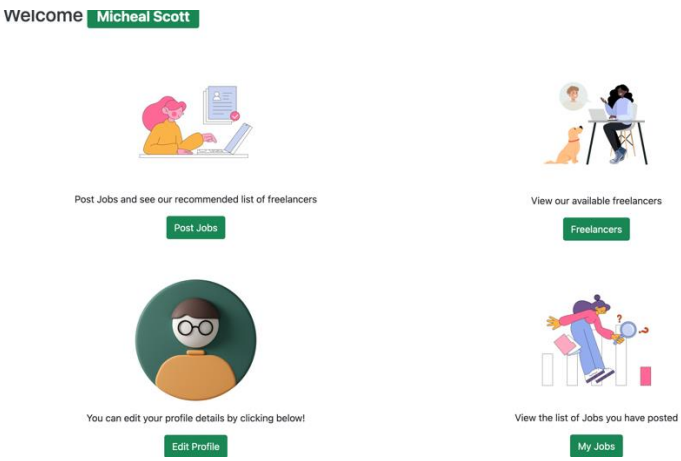


Figure [45]. *Business Dashboard*

Edit profile functionality works the same way for both the freelancers and business. The implementation is explained in freelancer dashboard section of front-end implementation.

View freelancer's functionality works the same way for both the freelancers and business. Added functionality for business is they can communicate with freelancers as they will have additional functionality to send email.

Post Jobs

This PostJobs.js component defines the user interface for the business to post new jobs. The author uses *useState()* which is a react hook that updates the state based on the user input. *OnSbumit ()* the author calls *addJob()* action which performs the HTTP request to register the user. This action is brought into the component by importing the action from the *job.js* action file and by using it as a *proptype*. Once the user fills in the necessary fields and clicks submit, the job gets added to the database as the action called sends the request to the server which then updates the database.

Once the job gets posted business gets navigated back to the dashboard and can view all the jobs posted by them by clicking view my jobs button.

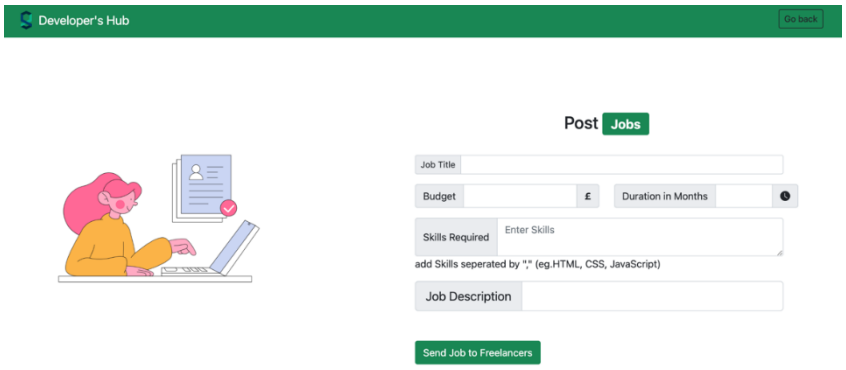


Figure [46]. *Post Jobs*

View my Jobs

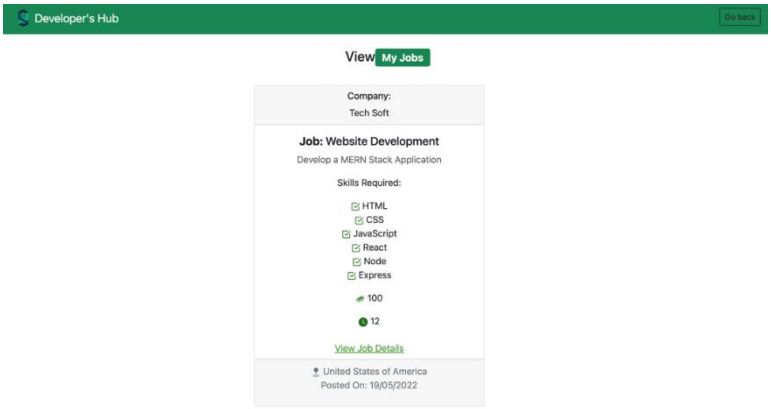
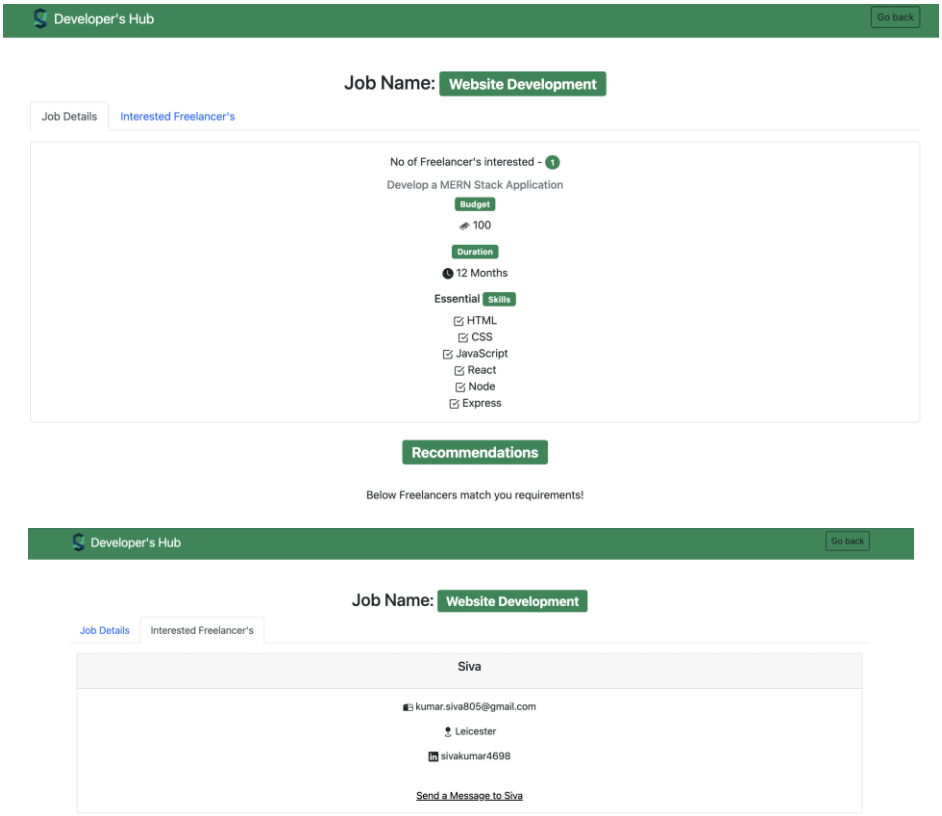


Figure [47]. *View my jobs list*

The user gets routed to *'/businessdashboard/myjobs'* which is defined as a route in app.js file. This route renders the component *<MyJobs />* which renders another component called *<MyJobView />.* The *<MyJobs />* component consists of jobs posted by business. whereas the *< MyJobView />* defines the interface of each job.

Once the business clicks my jobs, the *< MyJobs />* component imports *getjobs()* action which sends the HTTP request to the server to get the list of all jobs post by the business. The *getjobs()* action is imported as *proptype* and called using the react *useEffect()* hook. The jobs posted by business appears in the redux state, which is then passed into the *< MyJobView />* component which creates an interface for each job and displays it.

The jobs posted by the business get displayed in my jobs page, and each job has a functionality to view job details. When the business clicks "View job details" a couple of actions are called to get the job details such as interested freelancers and recommendations for the job.



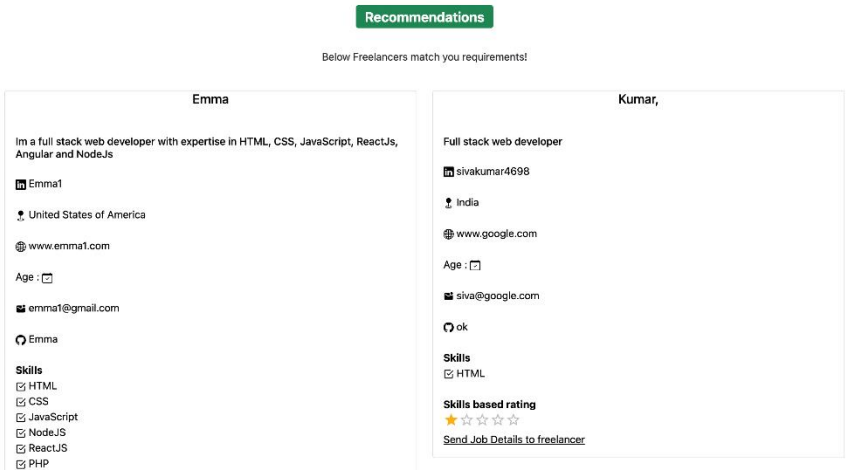


Figure [48]. *View job details and recommendations*

When the business clicks “send job details to freelancers” the job details will be sent to freelancer through email.

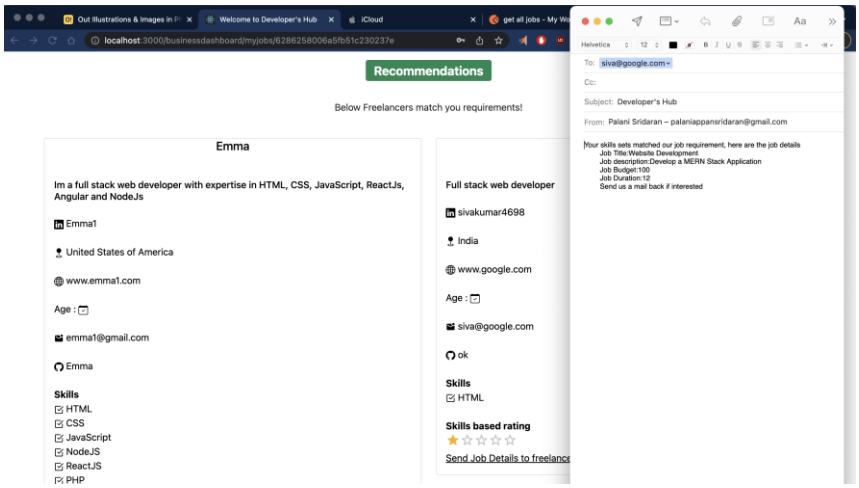


Figure [49]. *Individual Job view to get recommendations and interested freelancers list*

Recommendations

There is no separate API Endpoint created in the server to display the list of recommended freelancers. Instead, the recommendations are brought in using the available actions and by updating the reducer and passing down the state to the component.

The recommendations are content-based, in content-based filtering the recommendations are based on the contents of the user requirements. By following this approach, we can recommend freelancers to business based on the skill set of the freelancer [3]. The idea of the application is to recommend freelancers to jobs based on their skill sets. When business post job, they can view the recommended freelancers who perfectly match their skill requirements. This is achieved using React redux.

When the business user clicks on “view job details” the necessary action calls are made using axios, which sends requests to the server to get details of all freelancers. The component *<Job />* is rendered which uses redux state and JavaScript logic to loop through all the freelancer's skill sets available in the redux state and recommends freelancers with skill sets that match the job requirement skillset.

Admin

The administrator of the application can view all the freelancer and business-related details. They can remove jobs, freelancer profile and business profile if necessary. The admin will have access to see the MongoDB Atlas cluster where he can view the collections in the database. The collections are structured based on profile, jobs and user details which provides the admin a UI to view and manage them.

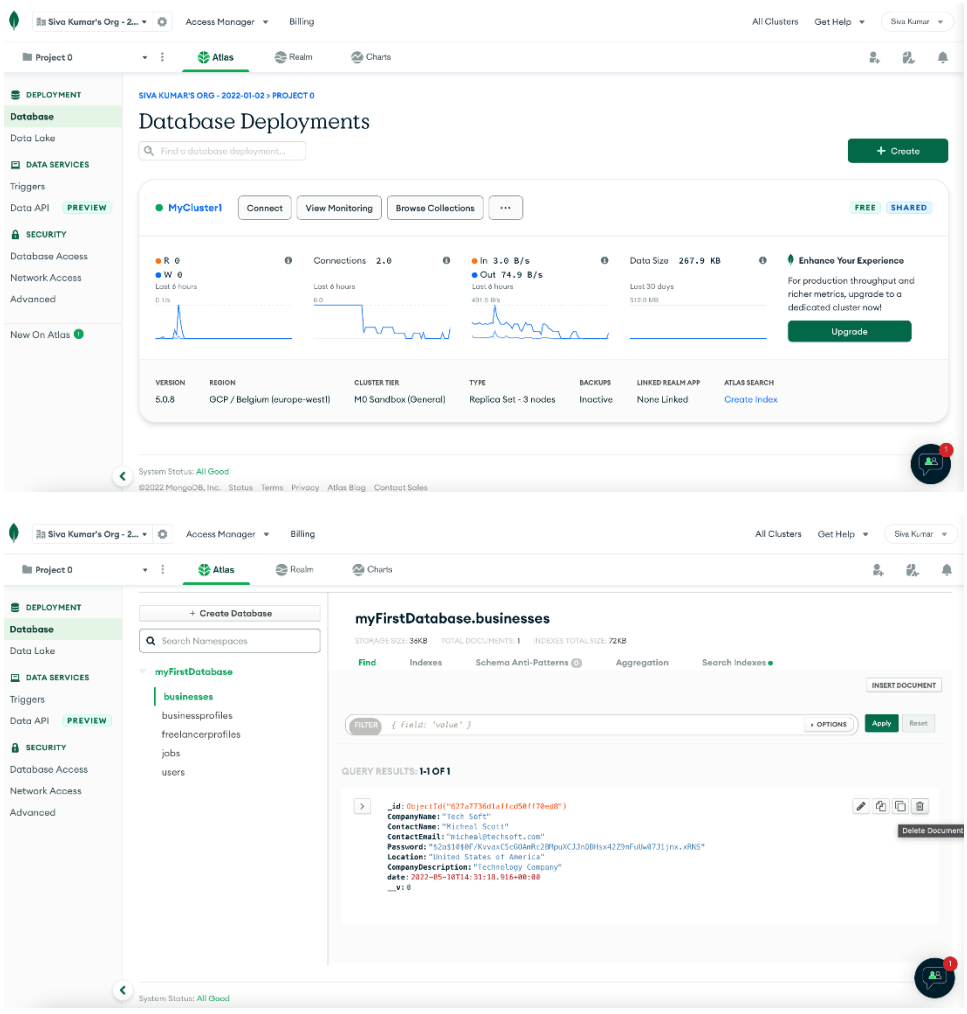


Figure [50]. *Admin interface to view and manage Profiles, jobs etc.*

Chapter – 7

Testing and Evaluation

An evaluation plan is to evaluate the achievements that have been accomplished from the actual aim.

Application Performance Evaluation

The aim is to evaluate the application based on its performance and operation. This will help us to determine the application performance against the objective/aim of the application and will help in determining where the app is missing its mark.

Outcome:

The author has performed an evaluation based on the application performance against the objective and the aim. The objective of the application was to provide a platform for businesses to find freelancers and for freelancers to find projects to work on. The additional aim was to bring in students to the application, all the key objectives have been achieved to the maximum extent. The author has put in the effort to learn and implement the application using multiple technologies. Which has also been proven successful.

User Experience Evaluation

To make the application successful we will have to benefit the user. Evaluating the user experience against the standards defined will ensure serving the purpose of the application.

Outcome:

The application functionalities are implemented based on the user story. This helps the user not to get lost when using the application. The user interactions are kept to the required extent which makes the user experience acceptable.

Heuristic Evaluation

This evaluation process will help in digging deep into the application's interface and identify issues related to usability. This will help us get an overview of the application and the problems that are affecting its usability.

Outcome:

The author has less experience designing websites UI, the author has given the best to make the interface usability user friendly. There are no problems that affect the usability of the application. The styling has not been up to the authors' expectations due to the limited timeline and slight lack in planning.

Functionality Evaluation

This evaluation process will help in determining whether the essential functionality has been implemented in the application.

Outcome:

All the user interactions are smooth, the functionalities are working as expected. The user will be able to view all the navigations clearly defined in each screen. The user navigation when performed with browser has some issues with http request. To overcome this issue the author has developed functionalities within the application where the user can perform navigation between screens by clicking the “Go back” button.

User Feedback Evaluation

Evaluation based on the user experience while using the application.

Outcome:

The author has shown the user interactions to the supervisor and has received good feedback. The author has also made other users use the application and the feedback received was reasonably good, although some users felt the input field requirements could have been further enhanced. The author has made note on the feedback and will improve more based on the feedback in future development.

Chapter - 8

Conclusion

The aim of this project is to develop a web application that connects businesses with freelancers using modern web development technologies. The author took the time to study each modern technology and learned the implementation process. The stack used for development is the MERN stack, all the concepts and implementation details used in each technology have been explained in detail in this paper. This documentation consists of all the essential information needed for implementing a MERN stack application.

The implementation of the requirements has been accomplished in the process of development. A completely functional website has been developed end to end. The application features the implementation of collaborating businesses and freelancers. When the business posts its requirements for development, they get a list of recommended freelancers who perfectly match its requirements. There are other features implemented such as interested freelancers, communication through email, freelancer profiles, ratings for freelancers based on their skills, and a dashboard for users. This application has unique features implemented that divide it from other similar applications.

Concluding Remarks

Although the implementation of the end-to-end web application has been successful. The application still has some drawbacks, and the functionality could have been further enhanced. The styling and design of the application on certain pages must be a drawback. The author has learned and implemented the design using react-bootstrap. The process was time-consuming, so the implementation has not been up to the mark. The technology stack used for implementation is a combination of most emerging business technologies nowadays. This project has given the author

the opportunity to learn all the technologies and combine them in developing a full-stack application.

References

Ado Kukic, S. V., 2021. *MongoDB & Mongoose: Compatibility and Comparison.* [Online]
Available at: https://www.mongodb.com/developer/article/mongoose-versus-nodejs-driver/

Anam Khan, A. Y. A. C. R. J. S., 2020. *Systematic Review on Recommendation System.* s.l.:ICACCCN.

Bootstrap, R., n.d. *react-bootstrap.* [Online]
Available at: https://react-bootstrap.github.io/

Denman, J., n.d. *WhatIs.* [Online]
Available at: https://www.techtarget.com/whatis/definition/Nodejs

Docs, M. W., n.d. *Mdn Web Docs.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started

Horiachko, A., 2021. *How to Choose the Best Technology Stack for Web Application Development: 10 Tips to Know.* [Online]
Available at: https://www.softermii.com/blog/10-tips-in-choosing-the-best-tech-stack-for-your-web-application#:~:text=A%20tech%20stack%20is%20a,side%20(back-end).

Majewski, M., 2019. *Top 6 Software Development Methodologies.* [Online]
Available at: https://blog.planview.com/top-6-software-development-methodologies/

MongoDB, n.d. *Fully Managed Database service.* [Online]
Available at:
https://www.mongodb.com/cloud/atlas/efficiency?utm_source=google&utm_campaign=gs_emea_united_kingdom_search_brand_atlas_desktop&utm_term=how%20to%20install%20mongodb&utm_medium=cpc_paid_search&utm_ad=e&utm_ad_campaign_id=1718986510&adgroup=80209772043&gcl

Mozilla, D., n.d. *Developer Mozilla.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Points, J. T., n.d. *React Redux.* [Online]
Available at: https://www.javatpoint.com/react-redux

Points, T., n.d. *Tutorial Points.* [Online]
Available at: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm

Rocca, B., 2019. *Introduction to recommender systems.* [Online]
Available at: https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada

Romero, G., 2021. *What is Postman API.* [Online]
Available at: https://www.encora.com/insights/what-is-postman-api-test

Wikipedia, n.d. *MongoDB.* [Online]
Available at: https://en.wikipedia.org/wiki/MongoDB