



UNIVERSITY OF  
**LEICESTER**

**DEPARTMENT OF INFORMATICS**

**CO7201 – MSc Individual Project**

**Project Title: Crypto Investment Assistant**

**Author: Raghu Ram Marni**

**Student Email: [rrcm1@student.le.ac.uk](mailto:rrcm1@student.le.ac.uk)**

**Student ID: 199048668**

**Project Supervisor: Victoria Wright**

**Second Marker: Dr. Yi Hong**

**Report Name: Dissertation**

**Word Count: 10054**

**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: Raghu Ram Marni

Date: 19-05-2022

## ABSTRACT

The modern web applications have come a long way in the evolution starting at pure texts with blue hyperlinks that would turn purple when clicked to robust web designs with pleasing user interfaces that are more secured, flexible, and scalable to meet the demand. JavaScript had contributed more to the web development than any language. It is the JavaScript's cut through the loading time as the code runs directly in the browser with client-side resultant logic gives the faster user experience. The asynchronous JavaScript helps in logical communication onto the server in the backend without disturbing the user interactivity that's going on in the front-end. The integration of HTML, CSS and JavaScript into a singular entity would result in a great responsive web design across various devices and its browsers. JavaScript on the server-side in form of Node JS is one of the robust frameworks for the backend development. The enhancement of functionalities in the Node JS using Express JS would be build efficient and complex-free programming to develop a robust API. This project named Crypto Investment Assistant is developed eloquently considering all the good aspects of various JavaScript frameworks, turning it into an advanced modern day web application. This project emphasises on the basic web functionalities like easy navigation and familiar UX as well as complex functionalities like SEO and robust UI. The fast routing between the pages and quick data fetching from the APIs to output the data in a user-friendly manner are the main goals to achieve in this project.

## Contents

<b>1. Introduction</b>	5
<b>2. Pre-requisites</b>	5
<b>3. Background and Requirements</b>	6
3.1 Motive and Background Work	6
3.2 Essential Requirements	7
3.3 Recommended Requirements	8
3.4 Optional Requirements	8
<b>4. Technical Specifications</b>	8
<b>5. Initial Project work</b>	10
Table 5.1 Initial Time-Plan	10
Table 5.2 Executed Time-Plan	11
5.3 Architectural Diagram	13
5.4 Use-case Diagram	15
<b>6. Front-end Development</b>	16
6.1 Development Packages used	22
<b>7. Front-end Components</b>	23
7.1 Login component	23
7.2 Signup component	24
7.3 Crypto component	26
7.4 GraphsPage component	27
7.5 Exchanges component	29
7.6 News component	32

7.7 Stocks component .....	34
7.8 Portfolio component .....	36
7.9 Search component .....	39
<b>8. Backend Development .....</b>	<b>41</b>
8.1 Data Schema .....	41
8.2 Middleware Setup .....	42
8.3 Backend Routing .....	43
8.4 Server-Side Rendering .....	46
8.5 Connecting Front-end with Back-end .....	48
8.6 Development Packages used .....	52
<b>9. Epilogue .....</b>	<b>53</b>
9.1 Testing .....	53
9.1.1 Functional Testing .....	53
9.1.2 Non-Functional Testing .....	56
9.2 Challenges faced .....	57
9.3 Future Development .....	58
<b>10. References .....</b>	<b>58</b>

## 1. Introduction

The existential crisis of financial markets and bodies during the 2008 Recession made many people around the world to lose faith in the banking and lending institutions. This gave rise to the new form of currency based on the transparent means of blockchain technology called the cryptocurrency. Bitcoin had been the kick-starter for cryptos and since then, cryptocurrencies had come a long way in establishing the functionality and usability to providing financial services to the people around the globe. The replacement of cryptos with the traditional financial institutions is another topic. The focus of this project is to enhance the day-to-day trading data, news, and exchanges. Keeping in mind about the hundreds of cryptos, real-time prices, speculative news, and huge chunks of crypto data, it would be a nightmare for a user like me to get an exact situation of the markets in seconds. This motivated me to develop a full-stack web application where, I as a user can access the real-time prices, exchange rates and news in one place. There are many challenges to develop a full-stack application like this, ranging from front-end designing, database management and server-side rendering. To overcome all the above challenges, this application is being developed using MERN stack (Mongo DB, Express JS, React JS, and Node JS).

## 2. Pre-requisites

As the full stack web development progresses rapidly, the requirement to learn new frameworks and technologies becomes stringent which in turn minimizes the development time and implement robust new features. But the basis of all these new frameworks lies within JavaScript, HTML and CSS. So, a great theoretical as well as practical knowledge on the basics would be a time saving effort for learning new frameworks. For this project, a good understanding of the blockchain, cryptocurrencies, crypto markets and stock markets would be a great added advantage in gathering functionalities based on the requirements. React JS supports this project in providing a full-fledged front-end functionality with the combination of JSX (JavaScript XML) that allows HTML in React, various UI libraries and dynamic web contents. The JavaScript runtime environment, Node JS enables this project with running the backend

frameworks within it. MongoDB provides the project with the non-relational NoSQL database that would be significant while working with large datasets and works like a charm within the Node environment. The use of the minimalistic and more flexible Node JS framework like Express JS for HTTP servers to handle numerous HTTP requests at a single URL would aid in the minimal time and effort in the backend development. So, we are combining all the four approaches to form a powerful stack of MERN in the development of this full stack web application. As this project needs a version control to store the code at various stages of development, knowledge on SVN (Subversion) is essential.

### 3. Background and Requirements

#### 3.1 Motive and Background Work

The retrospective nature of the human mind has brought us to the place where we are now (in a positive way). The ability to remember and recreate the events or situations had created robust developments after multiple attempts. The most common aspect of this irrelevant topic with this project is the past data. The study of data plays a key role in the prediction of the future markets at a minimal deviation from the result. This is very crucial in the most volatile and unpredictable markets like cryptocurrencies and stock markets to some extent. With that point in mind and having a short working experience with the web development, I came up an idea to build a full stack web application for the cryptos and ensuring to use as many web development concepts as possible to provide various functionalities to the application. But the knowledge of cryptos and the functioning of the financial markets is crucial in providing the user requirements as well as to cover broader user base. With the above requirement aspects and advanced web technologies, the developer had to go through extensive background work before the start of the development. The list of the background research done ranges from published papers to modern web development online courses and are mentioned below.

- Giancarlo Giudici, Alistair Milne & Dmitri Vinogradov 's Cryptocurrencies: market analysis and perspectives.

Available at: <https://link.springer.com/article/10.1007/s40812-019-00138-6> (Accessed: Week-1 14 February 2022).

- Min Xu, Xingtong Chen & Gang Kou 's A systematic review of blockchain.

Available at:

<https://jfin-swufe.springeropen.com/articles/10.1186/s40854-019-0147-z> (Accessed: Week-1 14 February 2022).

- Fran Casino, Thomas K. Dasaklis, and Constantinos Patsakis 's A systematic literature review of blockchain-based applications: Current status, classification, and open issues.

Available at:

<https://www.sciencedirect.com/science/article/pii/S0736585318306324> (Accessed: Week-1 16 February 2022).

- Sheikh Mohammad Idrees, Mariusz Nowostawski, Roshan Jameel and Ashish Kumar Mourya 's Security Aspects of Blockchain Technology Intended for Industrial Applications.

Available at:

<https://www.mdpi.com/2079-9292/10/8/951/htm> (Accessed: Week-1 17 February 2022).

- [Beginning Modern JavaScript \(Includes 10 Real Projects\) | Udemy](#) (Accessed: January 2022).
- [React Front To Back 2022 | Udemy](#) (Accessed: February 2022).

### 3.2 Essential Requirements

- User Authentication - For user registration and login
- Portfolio - For user's Crypto, Stock and Exchange Lookouts
- Cryptocurrencies - For list of crypto currencies and their respective data with good statistical GUI.
- Exchanges - For list of crypto exchanges based on their rankings and redirected links.
- News - For the latest news based on the crypto coins and markets around the world.
- Database (Mongo DB) - For the storage of user data.
- CSS library (Tailwind CSS) – Utility first CSS framework for low-level custom designs within HTML file.



### 3.3 Recommended Requirements

- Server-side rendering (Express JS) – Handling of HTTP server request, for the search engine optimization (SEO) and quick render of large JS files compared with single page applications (SPA).
- React UI library (Material UI) - For great graphical user interface.
- State Management Tools – React Hooks.
- Adding Stock data - To make it more functional financial assistant, adding various stock prices and show it in more graphical and statistical manner.

### 3.4 Optional Requirements

- Use of Loading templates.
- Providing dark mode for the whole web application.
- Providing a share option for the user to share a specific crypto to their friends through social media.

## 4. Technical Specifications

The following table clearly demonstrates the various tools, software and languages that are used to develop this full stack web application.

Type	Name	Version Used	Details
Operating System (optional)	Windows	11	The default PC operating system for development.
Programming Language	JavaScript	ES13	Providing functionalities for the web content in the pages.
	HTML	5.0	Providing the structure to the web pages and its contents.
	CSS	4.15	Providing the styles and layouts to web pages

Framework	React	17.0	JavaScript Library for building front-end user interfaces
	Express	4.17.3	A prebuilt Node JS framework for developing server-side applications along with server-side rendering
Runtime env	Node.js	17.9.0	An open-source server environment to run JavaScript on the server.
IDE	Visual Studio Code	1.66.1	The most popular and sought-after code editor along with complete development operations environment.
Database	Mongo DB	4.4	The highly available and scalable no SQL non-relational database.
Database Object Model tool	Mongoose	6.2.10	An object Document Modelling (ODM) for a non-relational database like MongoDB.
Database on cloud	MongoDB Atlas	---	A global cloud database as service for MongoDB.
Version Control	Tortoise SVN	1.10.5	The subversion client for managing different versions of the source code.

## 5. Initial Project work

Project of this size should be pre-planned before the initialization. So, the developer had come up with the following time-plan which is presented below on table 5.1 but executed all the plans within the timeline in table 5.2.

Table 5.1 Initial Time-Plan

Activity Name	Activity Start Date	Activity Commencement Week
Project Title	14-02-2022	Week 0
Background Research	14-02-2022	Week 0
Project Description	14-02-2022	Week 0
Front End Development Start	28-02-2022	Week 2
Web Designing & Data Fetching	28-02-2022	Week 2
Preliminary Report	01-03-2022	Week 2
Graphical Data Representation	07-03-2022	Week 3
State Management Implementation	14-03-2022	Week 4
Back End Development Start	21-03-2022	Week 5
Database Creation (MongoDB)	21-03-2022	Week 5
Interim Report	28-03-2022	Week 6
User Data Functional Implementation	28-03-2022	Week 6
Server-side Rendering	04-04-2022	Week 7
Optional Requirement Implementation	11-04-2022	Week 8
Dissertation	18-04-2022	Week 9
Running Test Cases	02-05-2022	Week 11
Project Submission	16-05-2022	Week 13

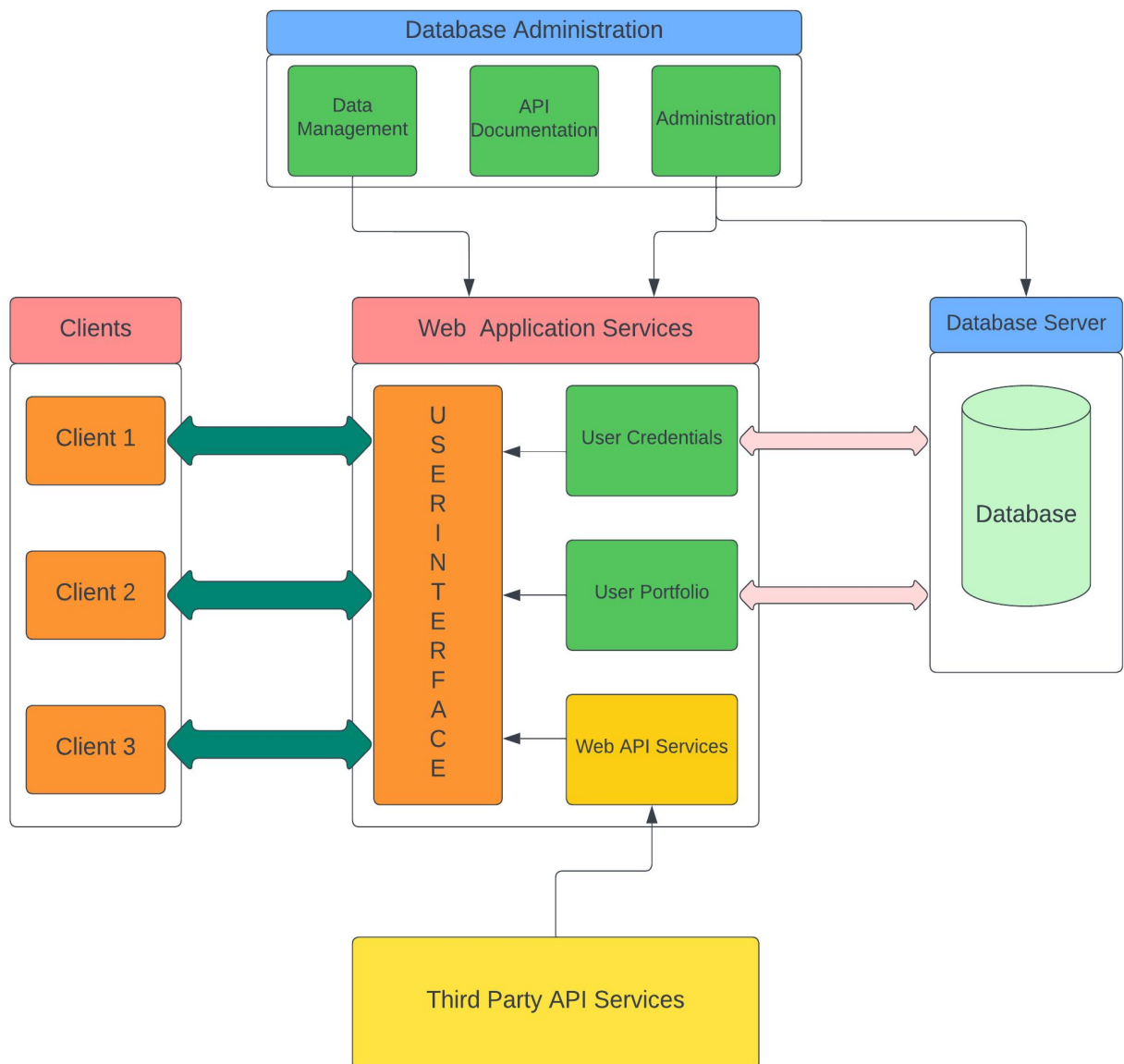
The initial time-plan had played a key role in shaping up the project as it is now. But, getting the targets to reach each milestone is a time consuming, effort putting process. There are some instances where, as a developer became hard to project the idea onto the output. Initially, creating most front-end components along with routing had been developed on time. The basic design part as well as providing graphical interface for each coin was a time-consuming process with lots of miss hits but reached them as planned. The change in time-plan had started with the backend development as the initial plan time frames for the backend are not practical to achieve. So, with delay on some of front-end components as well as delay in the creation of database had moved most of the activities in the initial time-plan to a week or more. But, leaving extra time in the initial time-plan for implementation of optional requirements had made it smooth and clear to use that period to implement all the essential and recommended requirements on time. The table below demonstrates more clearly about the difference between the intended initial time-plan and the executed time-plan.

Table 5.2 Executed Time-Plan

Activity Name	Activity Start Date	Activity Commencement Week
Project Title	14-02-2022	Week 0
Background Research	14-02-2022	Week 0
Project Description	14-02-2022	Week 0
Front End Development Start	28-02-2022	Week 2
Web Designing & Data Fetching	28-02-2022	Week 2
Preliminary Report	01-03-2022	Week 2
Graphical Data Representation	14-03-2022	Week 4
State Management Implementation	28-03-2022	Week 6
Back End Development Start	28-03-2022	Week 6

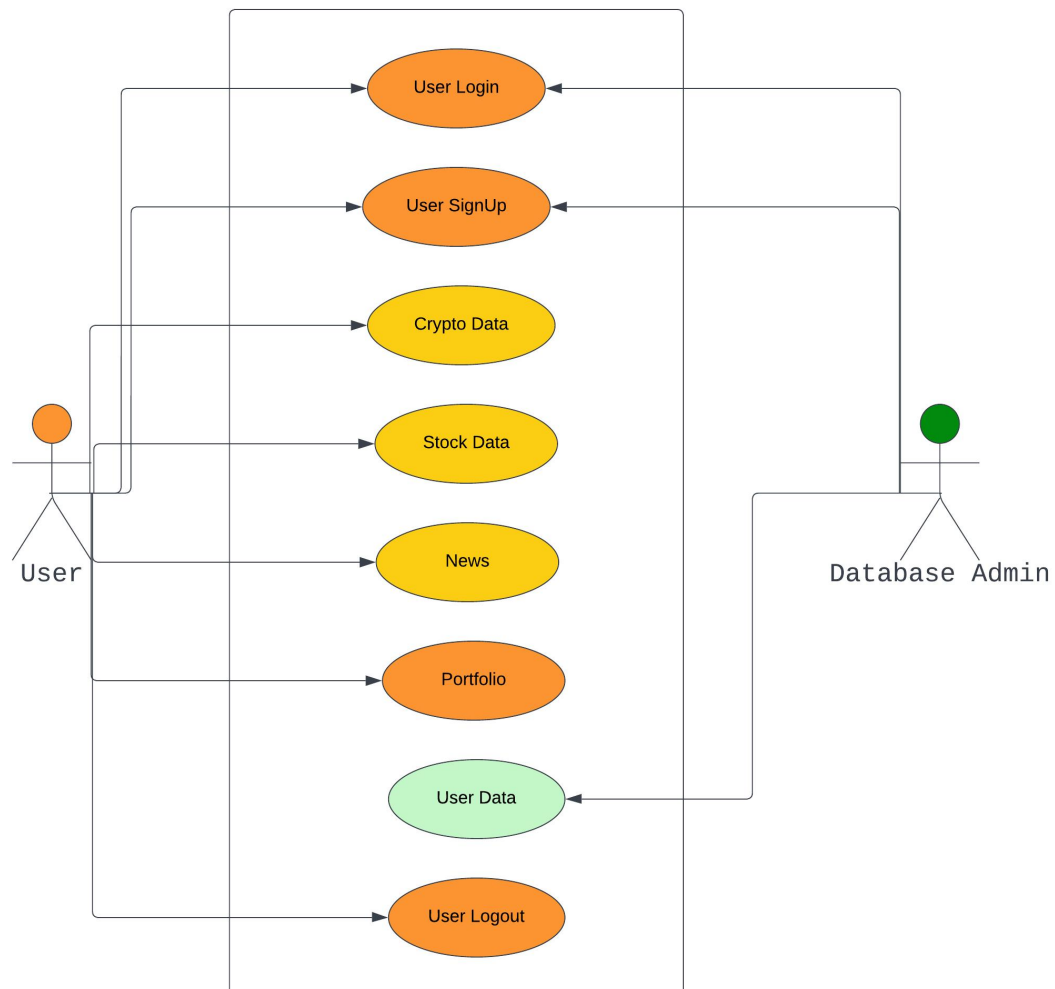
Database Creation (MongoDB)	28-03-2022	Week 6
Interim Report	28-03-2022	Week 6
User Data Functional Implementation	04-04-2022	Week 7
Server-side Rendering	11-04-2022	Week 8
Optional Requirement Implementation	18-04-2022	Week 9
Dissertation	18-04-2022	Week 9
Running Test Cases	09-05-2022	Week 12
Project Submission	19-05-2022	Week 13

## 5.3 Architectural Diagram



Broadly, the architecture of this project is divided into five parts. The user or the client faces the front-end of the architecture. The web application services consist of the user interface for the client to access the data and outputs, the web API services for gathering the real-time data from the third-party APIs through fetching, user credentials and portfolio for storing the user data on the backend server. Here, the database server is the real-time cloud database that is linked through MongoDB on the backend. The administration and management of the entire project is dependent on the database administration part which also consists of the API documentation for fetching the real-time data. The third-party API module plays a crucial role in providing the useful, reliable, and real-time data for the application to run relentlessly. The flow of data between the server and the web application services is very important in providing the users with their login and viewing of their portfolio. The web API services run parallelly with the data management of the application where both have a similar outcome with a different working mechanism. The possibility of using the application simultaneously by various clients at a given time is achieved by using the cloud database because of its flexibility, affordability, and scalability of managing database. So, the simple architecture diagram is essential in embodiment of working principle of the project as a whole. It also helps in analysing the scope of expandability in the near future.

## 5.4 Use-case Diagram



The use-case diagram emphasizes on the description of the high-level functionality of the application along with its scope. The interactions between the actors and the use cases help in determining the application's functionality along with the way the actors handle the application functions internally. The above is a simple use-case diagram that has two actors and eight functionalities. The user actor has the access and scope to use seven out of the eight functions but user login or user signup being the crucial one in accessing the remaining functions. The database admin has the real-time data access to the users' data along with their relative functionalities. The user's



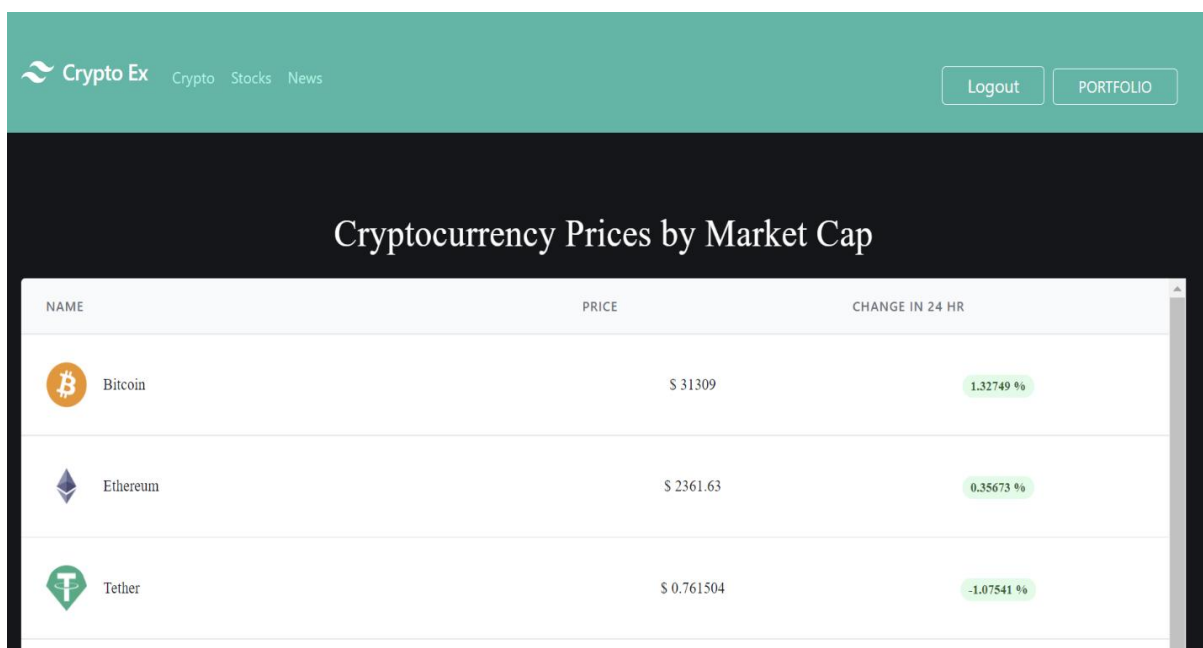
usage is pretty much limited to the accessing of various pages along with their personal portfolio page. But the database admin has more important role in attaching the user data to the cloud database as well as possessing the API documentation in fetching the real-time API data to the respective pages which are used by the user.

## 6. Front-end Development




Before getting started with the client-side development through React, a clear understanding of React life-cycle methods for the functional components is crucial. This demonstrates the working of the React components in an under-level format. Each React component is basically goes through three phases of its lifecycle i.e., mounting, updating, and unmounting. The process of React component rendering during the initial load is called mounting. It is basically an initialization to an instance of the React component. The change in the output as well as the instance of the initialized React component during the mount is called as updating. The process of updating can be carried out either by using props or through managing the state. These two ways of updating the state along with the hooks play a crucial role in the front-end development of this application. The hiding or the unshown of the mounted or updated React component or the instance of it when it is no longer needed is called unmounting. This is used deliberately for not showing the instance or the component after its use and thus ending the lifecycle of that particular React component or instance.

```
const [stocks, setStocks] = useState([]);
const [loading, setLoading] = useState(false);
console.log(stocks)
useEffect(() => {
  if(localStorage.getItem('token')){
    setLoading(true)
    axios.get(`https://api.coingecko.com/api/v3/coins/markets?vs_currency=us&ids=bitcoin,ethereum,cardano,binancecoin,ripple`)
      .then(result => {
        setStocks(result.data)
      })
    setLoading(false)
  } else {
    navigate('/Login')
  }
}, [])
```

From the above code snippet, the first instance to discuss is about the stocks. Here, stocks have been getting through updating lifecycle in the real-time through Coingecko API only when the `useState` has been initialized with the `setStocks()` method along with the `useEffect` that has been used to run at a particular moment after the mounting of the component. Alternatively, there has been another state for loading where it is run only during the fetching of the real-time data and gets unmounted whenever the updating of stocks data completes. In this way, by using multiple states as well as props we run all the lifecycle models that would be dependent with each other.



The screenshot shows the 'Crypto Ex' website with a teal header. The navigation bar includes 'Crypto', 'Stocks', and 'News'. On the right, there are 'Logout' and 'PORTFOLIO' buttons. The main heading is 'Cryptocurrency Prices by Market Cap'. Below this is a table with three columns: 'NAME', 'PRICE', and 'CHANGE IN 24 HR'.

NAME	PRICE	CHANGE IN 24 HR
 Bitcoin	\$ 31309	1.32749 %
 Ethereum	\$ 2361.63	0.35673 %
 Tether	\$ 0.761504	-1.07541 %

Before we start working with our React components, having a web design based on the utility first approach is obviously the better way to design this project. So, Tailwind CSS have been used for the outline as well as the component design based on each utility. The following two code snippets along with output would clarify the choice of this CSS framework.

```
<nav className="flex items-center justify-between flex-wrap bg-teal-500 p-6">
  <Link to={'/'}>
    <div className="flex items-center flex-shrink-0 text-white mr-6">
      <svg
        className="fill-current h-8 w-8 mr-2"
        width="54"
        height="54"
        viewBox="0 0 54 54"
        xmlns="http://www.w3.org/2000/svg">
        <path d="M13.5 22.1c1.8-7.2 6.3-10.8 13.5-10.8 10.8 0 12.15 8.1 17.55 9.45 3.6.9 6.75-.45 9.45-4.05-1.8 7.2-6.3 10.8-13.5 10.8-10.8 0-12.15 8.1-17.55 9.45 3.6.9 6.75-.45 9.45-4.05z"/>
      </svg>
      <span className="font-semibold text-xl tracking-tight">
        Crypto Ex
      </span>
    </div>
  </Link>
  {localStorage.getItem("token") ? (<
    <div className="block lg:hidden">
      <button className="flex items-center px-3 py-2 border rounded text-teal-200 border-teal-400 hover:text-white">
        <svg
          className="fill-current h-3 w-3"
          viewBox="0 0 20 20"
          xmlns="http://www.w3.org/2000/svg">
            <title>Home</title>
```

The static HeaderMenu component i.e., Navigation bar is present at any instance in the application. It acts like a static routing directory between all the main components. The styling for this component had been based on the utility first approach and also it supports the conditional JWT (JSON Web Token) where the main routing components are hidden when the user is logged out.

The below code snippet demonstrates about the linear development approach for the creation of the table for crypto data to have their heading for each column i.e., name, price and change in 24 hours. In this manner, Tailwind CSS is a powerful and fun framework to use for the utility first approach for the static as well as the dynamic React components.

```

<div className="shadow overflow-hidden border-b border-gray-200 sm:rounded-lg">
  <table className="min-w-full divide-y divide-gray-200">
    <thead className="bg-gray-50">
      <tr>
        <th
          scope="col"
          className="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-wider">
          Name
        </th>
        <th
          scope="col"
          className="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-wider">
          Price
        </th>
        <th
          scope="col"
          className="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-wider">
          Change in 24 hr
        </th>
      </tr>
    </thead>
    <tbody className="bg-white divide-y divide-gray-200">

```

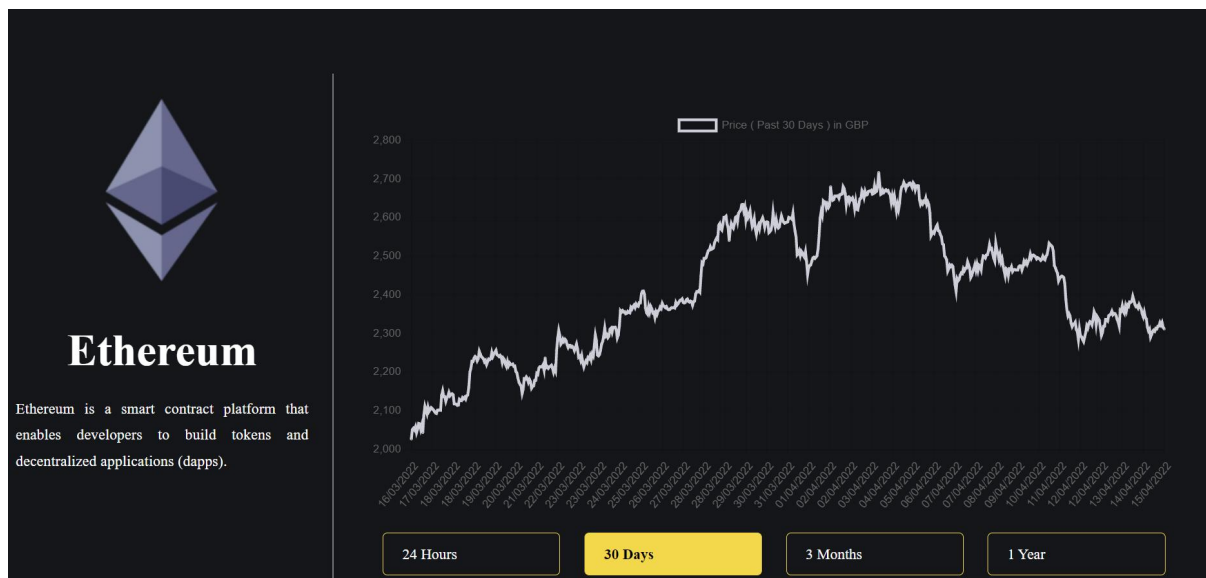
The approach of outputting the dynamic data after rendering it from an API would be more efficient by using a React UI library than Tailwind CSS. So, a popular React UI library called MUI (formerly Material UI) is used to handle and output the dynamic data. Various MUI components like LinearProgress, makeStyles, Typography, createTheme, CircularProgress and ThemeProvider are used in outputting the data that's been received from an API like Coingecko or Bing News.

```

<ThemeProvider theme={darkTheme}>
  <div className={classes.graph}>
    {!historicData || flag === false ? (
      <CircularProgress
        style={{ color: "gold" }}
        size={250}
        thickness={1}
      />
    ) : (
      <>
        <Line
          data={{
            labels: historicData.map((stock) => {
              let date = new Date(stock[0]);
              let time =
                date.getHours() > 12
                  ? `${date.getHours() - 12}:${date.getMinutes()} PM`
                  : `${date.getHours()}:${date.getMinutes()} AM`;
              return days === 1 ? time : date.toLocaleDateString();
            }),
            datasets: [
              {
                data: historicData.map((stock) => stock[1]),
                label: `Price ( Past ${days} Days ) in GBP`,
                borderColor: "#CFD0DC",
              },
            ],
          }}
        />
      </>
    )}
  </div>

```

From the above code snippet, the `CircularProgress` is used to output the loading template during the data retrieval, and it can be styled according to our choices like above. The dark theme for the application is also achieved through MUI's `ThemeProvider` component.



Another important aspect of the front-end development of this application is the use of a npm package for React called as `Chart.js` for outputting the data in the form of graphical interface. The wrapper of `chart.js` in React is `react-chartjs-2` that comes with a ton of customizable features like bar chart, line graph and pie charts. The above code snippet for outputting the data of Ethereum is outputted in the form of line graph through `chart.js`. So, the combinational use of Tailwind CSS, MUI and `chart.js` completes the client side designing part of the application.

The functionality part of the client side of the application depends heavily on React based on JavaScript. Various React concepts like state, props, state management, hooks, lifecycle methods and Context API are used in providing the responsiveness to the designed components.

```
const options = {
  method: 'GET',
  url: 'https://bing-news-search1.p.rapidapi.com/news/search?q=Cryptocurrency',
  params: { safeSearch: 'Off', textFormat: 'Raw', freshness: 'Day' },
  headers: {
    'X-BingApis-SDK': 'true',
    'X-RapidAPI-Host': 'bing-news-search1.p.rapidapi.com',
    'X-RapidAPI-Key': '70becd433cmshb2a121f67dbb62dp1dbb8bjsn5ae3bc519893',
  },
};

const initialState = {
  value: [],
};

export default function News() {
  const [data, setData] = useState(initialState);
  const [isFetching, setIsFetching] = useState(true);

  useEffect(() => {
    axios
      .request(options)
      .then(response => setData(response.data))
      .catch(error => console.error(error))
      .finally(() => setIsFetching(false));
  }, []);
```

The code snippet above clearly demonstrates the use of various React concepts. A JavaScript library called Axios is used to make HTTP requests instead of pre-existed Fetch as it is having the capability of single step process when dealing with JSON data whereas the later uses a two-step process namely making an initial request as well as using .json() method for the response to fetch the original data object. The project is completely built on the functional components rather than the class components in React. This approach underwhelmingly guarantees us with a use of much modern React concepts that are ranging from hooks and Context API. So, to manage the state in the above component, useState hook has been imported and two methods of setData() and setIsFetching() are used for two different instances of managing the state. Here, data is the current dataset that's being fetched from the Bing News API and isFetching is the loading conditional value for the fetching process. The useEffect hook provides us with running different instances of the useState hooks simultaneously after every render. It is basically providing React with the instructions to do after render.

## 6.1 Development Packages used

The packages play a crucial role in minimizing the time of the development. A package is basically an ordered library that can be downloaded through NPM(Node Package Manager). A library is a set of modules that contains the pre-existing JS code. React JS possess a huge collection of packages. Some of the packages by NPM that are used for the front-end development are mentioned below.

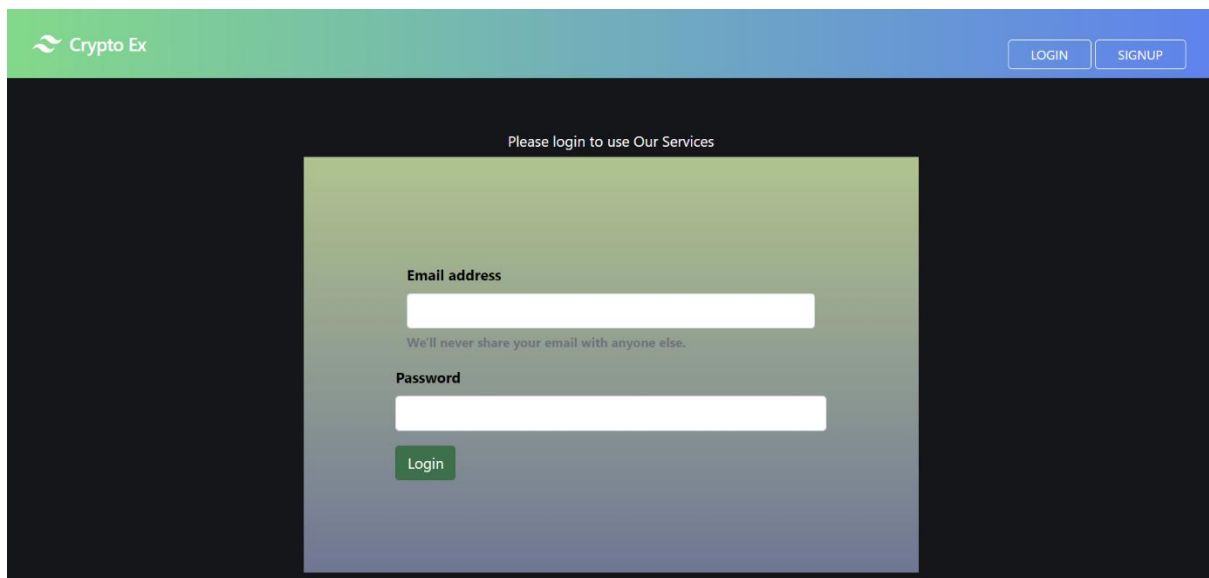
- i. [PostCSS](#) – Provides the linters through JS plugins for managing styles and CSS syntax.
- ii. [Autoprefixer](#) – A PostCSS extension tool to add vendor prefixes to the code so that our CSS code can run anywhere irrespective of the browsers.
- iii. [tailwindcss](#) – The highly customizable, low-level CSS framework for the utility first approach to build customized user interfaces in an effective way.
- iv. [React Router DOM](#) – An NPM package for the implementation of the dynamic routes and navigation between pages for the React application.
- v. [axios](#) – A popular NPM package for making HTTP requests from the external sources and returning data in form of promises.
- vi. [chart.js](#) – The JavaScript data visualisation library for creating charts and graphs.
- vii. [buffer](#) – Buffer deals with the binary data for a particular location in the memory. This helps to handle the streams of binary data.
- viii. [react-chartjs-2](#) – A popular library for Chart.js to use for the React components.
- ix. [react-paginate](#) – A simple React JS component to manage the pagination for tabled data in a particular page. It just gives the outline to divide the page, but we should add the functionalities according to our requirements.
- x. [react-share](#) – A popular React JS component to add the sharing social media icons as well as to provide external redirection to their respective websites whenever an icon is clicked but we should initially input title, URL, quotes, and styling.



## 7. Front-end Components

The project consists mainly 9 major React components as the intended outputs for the client-side projection but also consists innumerable supporting or adjective components for those major ones.

### 7.1 Login component



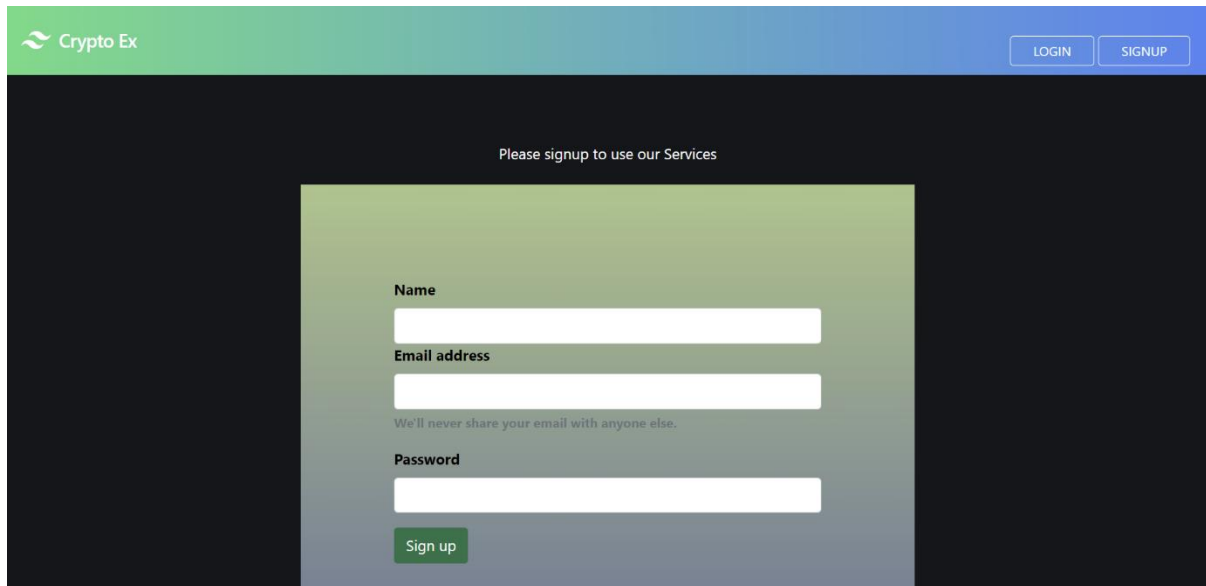
The screenshot displays the login interface for 'Crypto Ex'. At the top, a green header bar contains the 'Crypto Ex' logo on the left and 'LOGIN' and 'SIGNUP' buttons on the right. The main content area has a dark background. Centered in this area is a light green rectangular box. Inside this box, the text 'Please login to use Our Services' is at the top. Below it are two input fields: 'Email address' and 'Password'. A small line of text, 'We'll never share your email with anyone else.', is positioned between the two input fields. At the bottom of the box is a green 'Login' button.



```
const Login = (props) => {  
  const navigate = useNavigate();  
  
  const handleSubmit = async (e) => {  
    e.preventDefault();  
    const response = await fetch("http://localhost:5000/api/auth/login", {  
      method: "POST",  
      headers: {  
        "Content-Type": "application/json",  
      },  
      body: JSON.stringify({  
        email: credentials.email,  
        password: credentials.password,  
      }),  
    });  
    const json = await response.json();  
    console.log(json);  
    if (json.success) {  
      props.showAlert("login successfully", "success");  
      //save the auth token and redirect back to  
      localStorage.setItem("token", json.authtoken);  
      navigate("/");  
    } else {  
      props.showAlert("invalid credentials", "danger");  
    }  
  };  
  
  const [credentials, setCredentials] = useState({ email: "", password: "" });  
  const onChange = (e) => {  
    setCredentials({ ...credentials, [e.target.name]: e.target.value });  
  };  
};
```

The login component is made up of a sign-in form that consists of two labels and a login button. After filling the form and clicking the login button, initialises a function called `handleSubmit()` that has been shown from the above code snippet. As the function is a promise, the `fetch` is used to post the filled form data with the backend API. Before sending it, the form data is converted to string using the method `JSON.stringify()`. As the function returns a promise, it takes time to get the response in form of JSON from the backend and it can satisfy two conditions. i.e., either the credential matches and navigate to `'/'` page or an invalid credentials conditional. After logging in, JSON authenticate token acts as credentials for the time being as the user starts routing between the pages and is stored in the local storage on temporary basis.

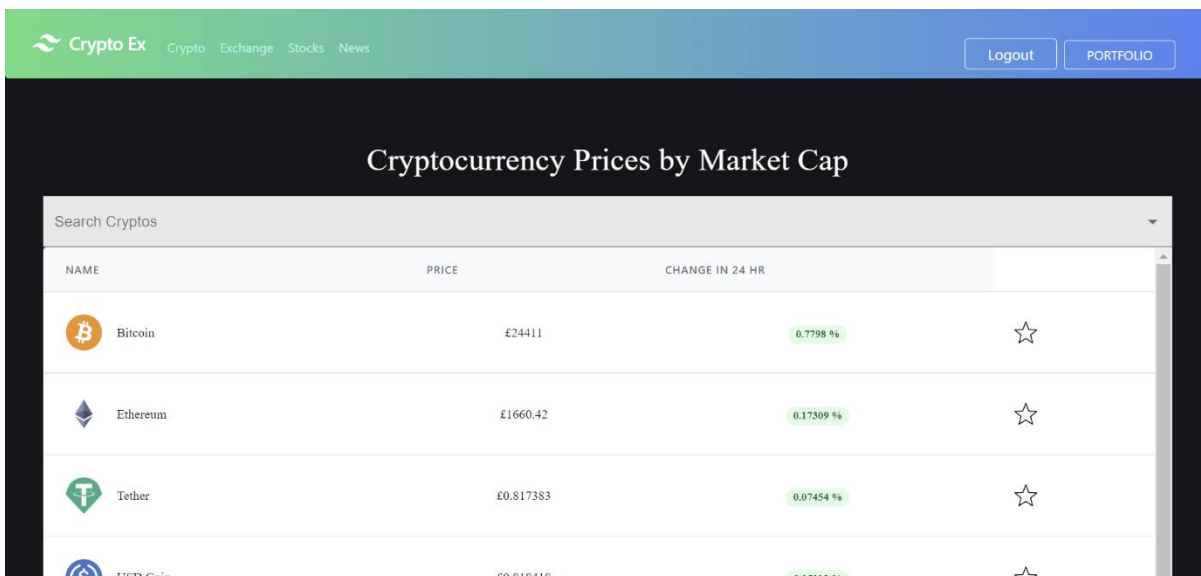
## 7.2 Signup component

The image shows a web interface for 'Crypto Ex'. At the top, there is a green header with the 'Crypto Ex' logo on the left and 'LOGIN' and 'SIGNUP' buttons on the right. The main content area has a dark background. In the center, there is a light green rectangular box containing a signup form. Above the form, the text 'Please signup to use our Services' is displayed. The form itself has three input fields: 'Name', 'Email address', and 'Password'. Below the 'Email address' field, there is a small line of text: 'We'll never share your email with anyone else.' At the bottom of the form is a green 'Sign up' button.





The signup component is made up of a form that consists of three labels and a signup button. The design principle and working mechanism of this component resembles that of the login component but with addition to extra features. Firstly, after the input data is send to the backend in form of JSON, there would be an alert to the new user if the email id already exists. Secondly, an id will be issued to the new user after the authentication token of the user created in the backend. This results the new user in accessing the application instantly after the signup without the login. There are some additional security features that are applied for the new user's data and will be discussed in the backend documentation.

```
const handleSubmit = async (e) => {  
  const response = await fetch("http://localhost:5000/api/auth/createuser", {  
    method: "POST",  
    headers: {  
      "Content-Type": "application/json",  
    },  
    body: JSON.stringify({  
      name: credentials.name,  
      email: credentials.email,  
      password: credentials.password,  
      cpassword: credentials.cpassword,  
    }),  
  });  
  const json = await response.json();  
  if (json.success) {  
    props.showAlert("account created successfully", "success");  
    //save the auth token and redirect back to  
    localStorage.setItem("token", json.authtoken);  
    localStorage.setItem("_id", json?.user?._id);  
    navigate("/");  
  } else {  
    props.showAlert("email already exist", "danger");  
  }  
};  
  
const [credentials, setCredentials] = useState({  
  name: "",  
  email: "",  
  password: "",  
});  
  
const onChange = (e) => {  
  setCredentials({ ...credentials, [e.target.name]: e.target.value });  
};
```

## 7.3 Crypto component



The screenshot shows the 'Crypto Ex' website interface. At the top, there's a navigation bar with 'Crypto', 'Exchange', 'Stocks', and 'News' links, along with 'Logout' and 'PORTFOLIO' buttons. The main heading is 'Cryptocurrency Prices by Market Cap'. Below this is a search bar labeled 'Search Cryptos'. The table displays the following data:

NAME	PRICE	CHANGE IN 24 HR	
 Bitcoin	£24411	0.7798 %	☆
 Ethereum	£1660.42	0.17309 %	☆
 Tether	£0.817383	0.07454 %	☆
 USD Coin	£0.818418	0.15113 %	☆

```

const cryptoPerPage = 10;
const Homepage = () => {
  const navigate = useNavigate();

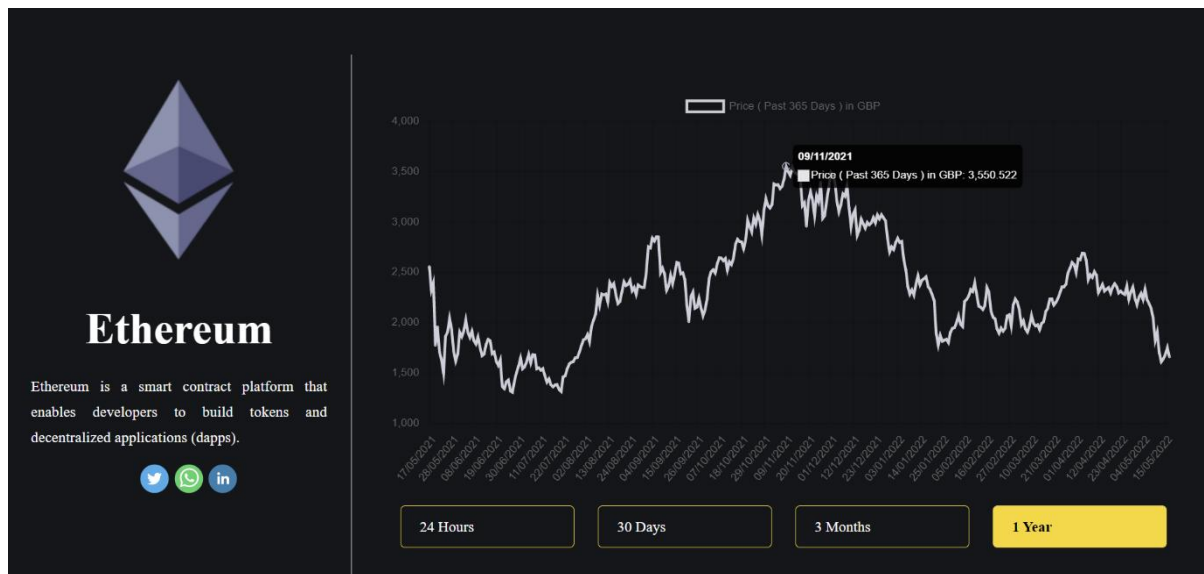
  const [stocks, setStocks] = useState([]);
  const [favourites, setFavourites] = useState(
    JSON.parse(localStorage.getItem("favourites")) ?? []
  );
  const [loading, setLoading] = useState(false);
  const [initialCrypto, setInitialCrypto] = useState(0);
  const [lastCrypto, setLastCrypto] = useState(10);

  console.log(stocks);
  useEffect(() => {
    if (localStorage.getItem("token")) {
      setLoading(true);
      axios
        .get(
          `https://api.coingecko.com/api/v3/coins/markets?vs_currency=gbp&order=market_cap_desc&per_page=100&page=1&sparkline=false`
        )
        .then((result) => {
          setStocks(result.data);
        });
      setLoading(false);
    } else {
      navigate("/Login");
    }
  }, []);
}

```

The crypto component is made up a list of cryptocurrencies that are showcased in descending order based on their market capitals. A third-party API called Coingecko provides with the real-time price of the crypto as well as 24-hour price change. As there would be state change more frequently, the useState hook is used to extract the change in state values and useEffect() hook is used to run a function if there would be change in the state of the component. So, useEffect hook helps in updating the prices whenever a refresh of the page takes place.

## 7.4 GraphsPage component



```
import { chartDays } from "../config/data";
import { Chart as ChartJS, registerables } from 'chart.js';
import { Chart, Line } from 'react-chartjs-2';
ChartJS.register(...registerables);
const Graphspage = () => {
  const {id} = useParams()
  const [stock, setStocks] = useState()
  const [days, setDays] = useState(1);
  const [historicData, setHistoricData] = useState();
  const [flag, setflag] = useState(false);

  useEffect(() => {
    axios.get(`https://api.coingecko.com/api/v3/coins/${id}`)
      .then(result => {
        setStocks(result.data)
      })
  }, [id])
  const fetchHistoricData = async () => {
    const { data } = await axios.get(`https://api.coingecko.com/api/v3/coins/${id}/market_chart?vs_currency=gbp&days=${days}`)
    setflag(true);
    setHistoricData(data.prices);
  };

  console.log(stock);

  useEffect(() => {
    fetchHistoricData();
  }, [days]);
}
```

The graphs page is the pictorial representation of each coin data in form of the graphs according to their respective timelines. A JavaScript data visualization library called chart.js is used to output the crypto prices in form of the graphs. The React features like `useState()` and `useEffect()` are used to access the real time data through changes in the state and updating the data after every page refresh. Here, four timelines are used ranging from the 24 hours to a year. The user has the option to select his desired timeline by just

toggling between them. Along with the coin data, the coin historical data has been fetched from the Coingecko API in real time through axios. The number of days acts as the dependency variable for the useEffect() hook. Whenever there will be change in the timeline through its button click by the user, the useEffect hook comes to the play and updates the state as well. There is also an additional feature for user to share their desired crypto with whomever they wish through Twitter, WhatsApp, or LinkedIn just by signing into their profiles and selecting the user to whom they want to send. So, their respective icons have been placed just below the coin description.

```
data={
  labels: historicData.map((stock) => {
    let date = new Date(stock[0]);
    let time =
      date.getHours() > 12
        ? `${date.getHours() - 12}:${date.getMinutes()} PM`
        : `${date.getHours()}:${date.getMinutes()} AM`;
    return days === 1 ? time : date.toLocaleDateString();
  }),
  datasets: [
    {
      data: historicData.map((stock) => stock[1]),
      label: `Price ( Past ${days} Days ) in GBP`,
      borderColor: "#CFD0DC",
    },
  ],
}
}}

{chartDays.map((day) => (
  <SelectButton
    key={day.value}
    onClick={() => {setDays(day.value);
      setflag(false);
    }}
    selected={day.value === days}
  >
    {day.label}
  </SelectButton>
))}
```

The above code snippet details about the logic that is used to fetch historical data with reference to the current date in the user's system. The historic data changes automatically whenever there is change in the local date string of the user. The second part of the code snippet elaborates about the change in the graph according to the days whenever the user toggle between the four-time constraints. The feature flag technique is implemented that reacts to the timelines whenever the user toggles between them.

## 7.5 Exchanges component

NAME	WEBSITE	COUNTRY	TRUST RANK
Binance	<a href="https://www.binance.com/">https://www.binance.com/</a>	Cayman Islands	1
FTX	<a href="https://ftx.com/">https://ftx.com/</a>	Antigua and Barbuda	2
Coinbase Exchange	<a href="https://www.coinbase.com">https://www.coinbase.com</a>	United States	3
OKX	<a href="https://www.okx.com">https://www.okx.com</a>	Belize	4

The crypto exchanges component's aim is to display and output the list of all the crypto exchanges according to their trust ranks as well as providing links to their respective websites whenever the user clicks them. Coming to the built of this component, all the exchanges data along with their home country, their URL and trust ranking is being received from the third-party API called CoinGecko through axios in real-time.

```
const ExchangePage = () => {
  const navigate= useNavigate();

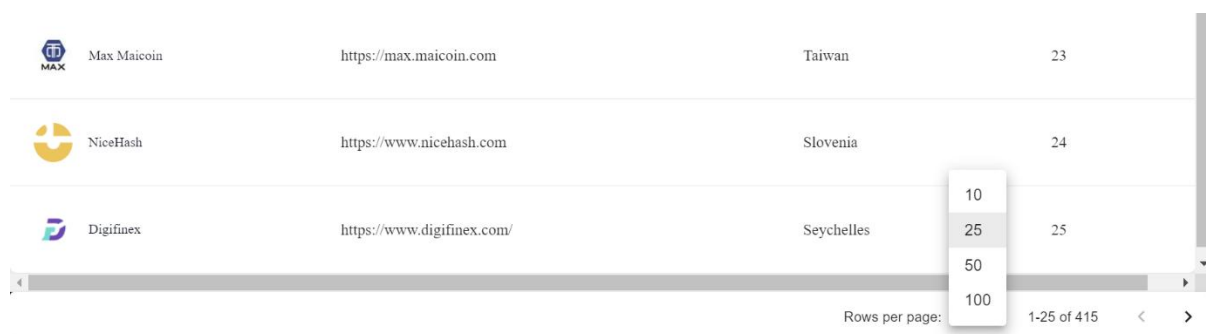
  const [exchanges, setExchanges] = useState([]);
  const [loading, setLoading] = useState(false);
  const [pagination, setPagination] = useState(null);
  const getData = (pageNumber, perPage=25) => {
    if(localStorage.getItem('token')){
      setLoading(true)
      axios.get( `https://api.coingecko.com/api/v3/exchanges?page=${pageNumber}&per_page=${perPage}` )
        .then(result => {
          setExchanges(result.data);
          setPagination({
            count: Number(result.headers['total']),
            rowsPerPage: Number(result.headers['per-page']),
            page:pageNumber
          });
        })
      setLoading(false)
    }
    else{
      navigate('/Login')
    }
  }




  useEffect(() => {
    getData(1);
  },[])
}
```

Initially, the state of three elements is defined by using useState() hook. The exchanges are an array of list data that is received from the API. The



loading comes into the play whenever there is get request from the user. The pagination helps in dividing the huge list of exchange data into minimal and navigable list according to the user's wish of selecting the number of exchanges per each page. The `useEffect()` hook becomes handy whenever there is change of state in the component and gets the refreshed real-time data. Prior to getting data from CoinGecko API, there is a JWT verification done by the server on the backend by accessing the stored token from the local storage of the browser. If the token doesn't match with the initial one, the current page is navigated back to login page by the built-in `useNavigate()` method for the user to verify with their credentials.



	Max Maicoín	<a href="https://max.maicoín.com">https://max.maicoín.com</a>	Taiwan	23
	NiceHash	<a href="https://www.nicehash.com">https://www.nicehash.com</a>	Slovenia	24
	Digifinex	<a href="https://www.digifinex.com/">https://www.digifinex.com/</a>	Seychelles	25

Rows per page: 10 25 50 100 1-25 of 415

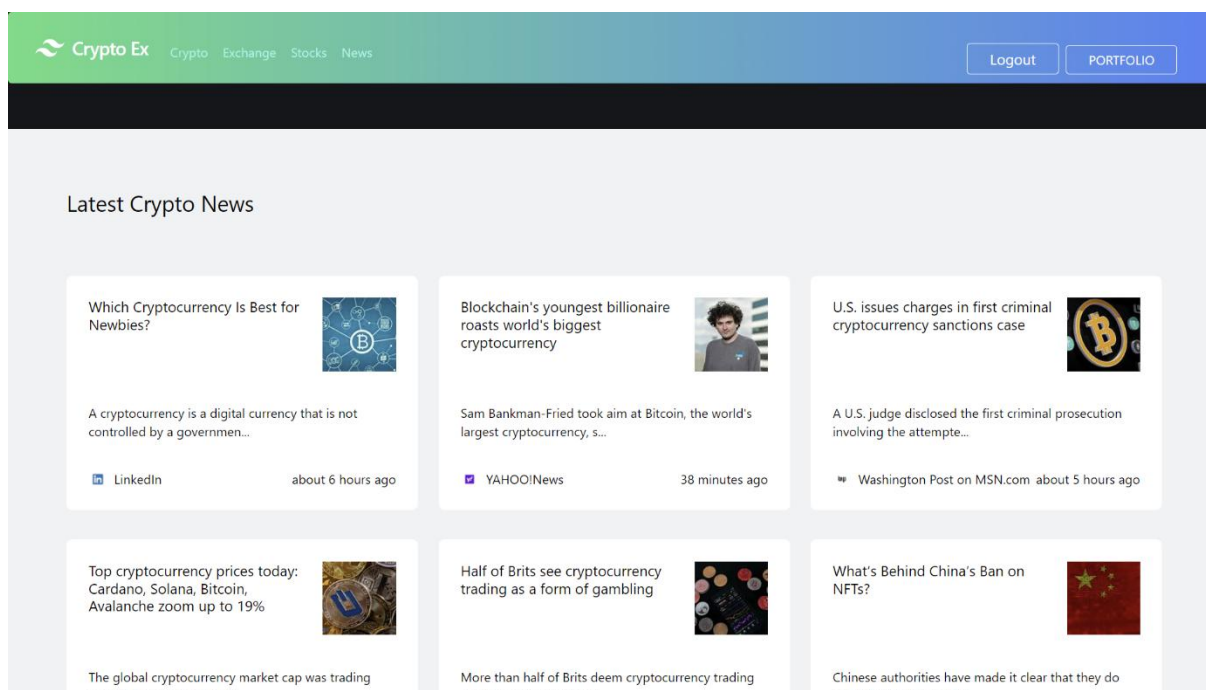
```
{pagination && <TablePagination
  component={Paper}
  count={pagination.count}
  page={pagination.page-1}
  rowsPerPage={pagination.rowsPerPage}
  onPageChange={(e, page) => {
    |   getData(page+1);
    | }}
  onRowsPerPageChange={(e) => {
    |   getData(pagination.page, e.target.value)
    | }}
></TablePagination>}
```

The above code implements the pagination for the exchanges component. Here, the count implies to the total number of exchanges and the page implies to a specific page with an id of its page number. The rowsPerPage implies to the specific user selection by toggling between the four values of the selection button. For this, the dynamic and highly customizable



TablePagination component of Material UI (MUI) was used, and all the input code was wrapped around it based on the props that were present in its docs. This whole concept of pagination is implemented at the footer section of the exchanges component.

## 7.6 News component



```
const options = {
  method: 'GET',
  url: 'https://bing-news-search1.p.rapidapi.com/news/search?q=Cryptocurrency',
  params: { safeSearch: 'Off', textFormat: 'Raw', freshness: 'Day' },
  headers: {
    'X-BingApis-SDK': 'true',
    'X-RapidAPI-Host': 'bing-news-search1.p.rapidapi.com',
    'X-RapidAPI-Key': '70becd433cmshb2a121f67dbb62dp1dbb8bj5n5ae3bc519893',
  },
};

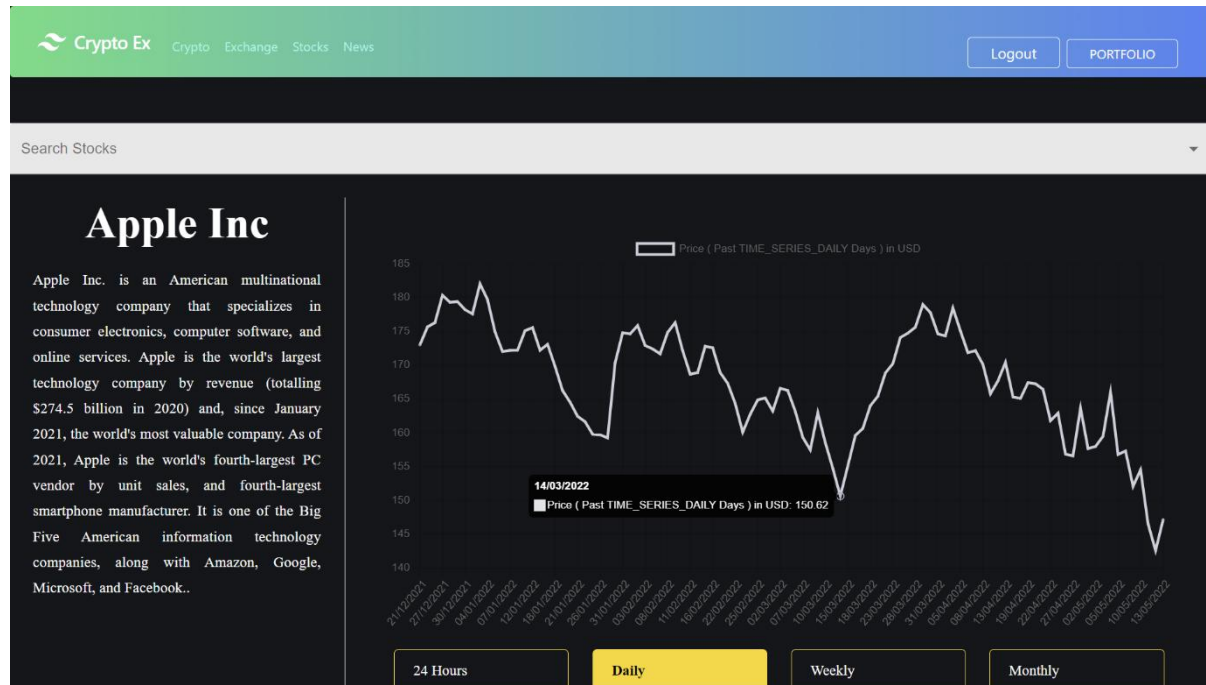
const initialState = {
  value: [],
};

export default function News() {
  const [data, setData] = useState(initialState);
  const [isFetching, setIsFetching] = useState(true);

  useEffect(() => {
    axios
      .request(options)
      .then(response => setData(response.data))
      .catch(error => console.error(error))
      .finally(() => setIsFetching(false));
  }, []);
```

The news component fetches the real time cryptocurrency news data by using Bing News API. Here, a third-party API library called Rapid API is used to connect with the Bing News API. This component has only two states, namely data state and fetching state. The setData() method updates the state every time there is change in the data and setIsFetching() method runs whenever there's been an updating in the news data. The useEffect hook is used to run whenever there's been a change of state in the virtual DOM of the React library. The output of only 10 latest news tiles on the news page is displayed and gets refreshed every time whenever the page reloads.

## 7.7 Stocks component



The stocks component deals with the output of the stock data of a particular company stock on the basis of their timeline. The user can get the stock data in a graphical user interface with a search of a particular stock in the search bar that is present between the navigation bar and the stocks container. The output is basically divided into two parts. The first part consists of a sidebar where the title of the stock as well as its historical description is depicted. The other half of the output showcases the value of the stock price according to the user input. There are four timelines to view the stock price of that particular stock at any given time. The 24 hours button just displays the change in the stock price in a graphical form for the past 24 hours of the operation of the stock exchange. In the stock market sense, this time period of 24 hours is called as intraday. The Daily button selection outputs all the recent stock prices and its changes in a graphical form from the last 6 months. The third selection of weekly timeline enables the graph to display all the weekly values of that particular stock from the initial time of its offering. The monthly timeline is similar to the weekly but just differs with the monthly stock price

since the inauguration of the stock on its exchange. The hovering of the mouse on a particular point on the graph gives the representation output of the date or time with the price of that particular stock at that particular instance.

```
import React, {useState,useEffect} from 'react';
import axios from 'axios';
import ReactHtmlParser from "react-html-parser";
import { LinearProgress, makeStyles, Typography, CircularProgress,
  createTheme,
  ThemeProvider, } from "@material-ui/core";
import SelectButton from "../SelectButton";

import { stockDays } from "../config/data";
import { Chart as ChartJS, registerables } from 'chart.js';
import { Chart , Line} from 'react-chartjs-2'
import SearchComponent from './SearchComponent';
ChartJS.register(...registerables);
const Stocks = (props) => {
  const [stock,setStocks] = useState()
  const [symbol,setSymbol] = useState("AAPL");
  const [days, setDays] = useState("TIME_SERIES_INTRADAY");
  const [resKey, setResKey] = useState("Time Series (5min)");
  const [historicData, setHistoricData] = useState();
  const [flag,setflag] = useState(false);

  const getDetails = async () => {
    axios.get( `http://localhost:5000/api/stocks/details?symbol=${symbol}` )
      .then(result => {
        setStocks(result.data)
      });
  }

  useEffect(() => {
    getDetails();
  },[symbol])

  const fetchHistoricData = async () => {
    const ( data ) = await axios.get( `http://localhost:5000/api/stocks/timeseries` )
    setflag(true);
    if(!data[resKey]) {
      props.showAlert(data.Note,"danger");
      setHistoricData([]);
    }
    else {
      const keys = Object.keys(data[resKey]);
      const result =[];
      for(let i=0;i<keys.length;i++) {
        result.push([keys[i], data[resKey][keys[i]]["4. close"]]);
      }
      setHistoricData(result);
    }
  }

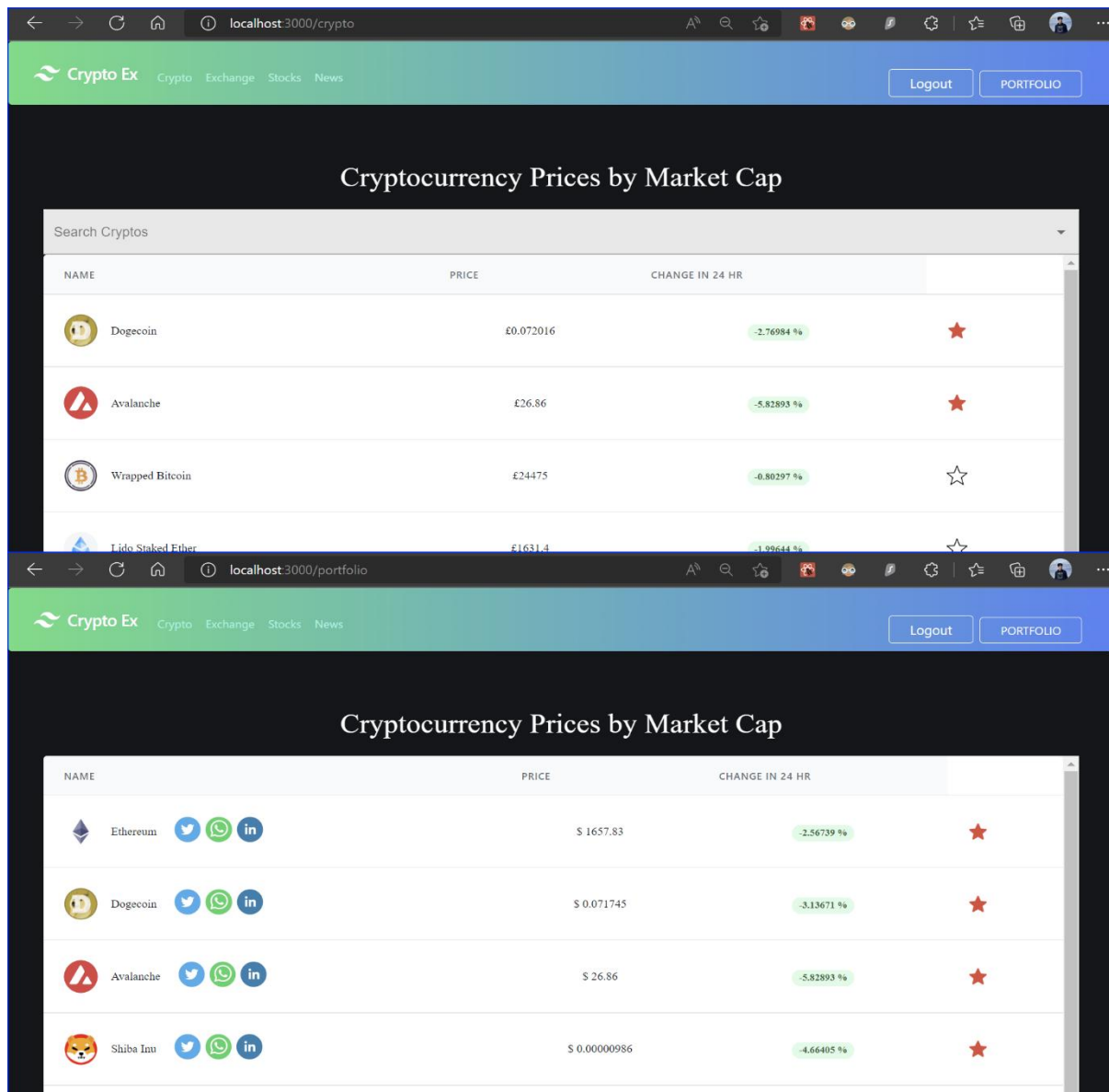
  console.log(stock);

  useEffect(() => {
    fetchHistoricData();
  }, [symbol, days]);
}
```

Most of the functions and states in the stock component are reused from the GraphsPage component. But it is only limited to the graphical representation of the stock data. It doesn't apply to the fetching of the real-time data from an external API on the frontend. Here, the stock data is fetched on the real-time basis on the backend which in turn would result to the server-side rendering of the stock data. Axios is used to fetch the stock data with respect to its stock symbol that is present on the backend API 'localhost:5000/api/stocks/'. There are in total five states to manage all the data and responses in this component. The setStocks() method is used to fetch the stock data of the resultant search value. Each stock has its own short form of its name through a symbol. So, whenever the user wants to search a stock, they should input the symbol of that particular stock where setSymbol() method is called to search that particular stock from the real-time database. The setDays(), setResKey() and setHistoricData() are all the methods that are used to divide the data output into four timelines according to the user's selection. The setDays() plays a key role in determining the number of days of each timeline according to their stock data. The setResKey() is the division of that timeline according to the requirement of the user to their needs. The

setHistoricData() is the managing of the collection of the stock data and outputting only the desired one according to the timeline. Whenever the stocks page is opened, the initial and default symbol is set for the Apple Inc (APPL) stock along with its intraday data with respect to its response time interval (resKey) as 5 minutes. The historicData comes into play whenever the user toggles to other timelines omitting the 24 hours timeline.

## 7.8 Portfolio component



The portfolio component enables the user with a feature to select their favourite cryptos. This in turn saves their time in searching their desired coins and gives a direct access to the graphs page of that specific coin whenever they want. There is also an additional feature for user to share their favourite crypto with whomever they wish through Twitter, WhatsApp, or LinkedIn just by signing into their profiles and selecting the user to whom they want to send. So, their respective icons have been placed just right to the coin name. Adding a

coin to their favourites in portfolio is just a tap away from clicking the star button that is placed to the right corner of every coin row.

```
export default function Portfolio() {
  const navigate = useNavigate();

  const [stocks, setStocks] = useState([]);
  const [favourites, setFavourites] = useState(
    JSON.parse(localStorage.getItem("favourites")) ?? []
  );
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    if (localStorage.getItem("token")) {
      setLoading(true);
      axios
        .get(
          `https://api.coingecko.com/api/v3/coins/markets?vs_currency=gbp`
        )
        .then((result) => {
          setStocks(result.data);
        });
      setLoading(false);
    } else {
      navigate("/Login");
    }
  }, []);

  const handleSubmit = async (e) => {
    if (favourites.includes(e)) {
      var fav = favourites.filter((d) => d !== e).map((d) => d);
    } else {
      fav = [...favourites, e];
    }

    const response = await fetch("http://localhost:5000/api/auth/update", {
      method: "PATCH",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        _id: localStorage.getItem("_id"),
        favourites: fav,
      }),
    });

    const json = await response.json();
    setFavourites(json.favourites);
    console.log(json);
    localStorage.setItem("favourites", JSON.stringify(json.favourites));
  };
}
```

\*Portfolio component side code

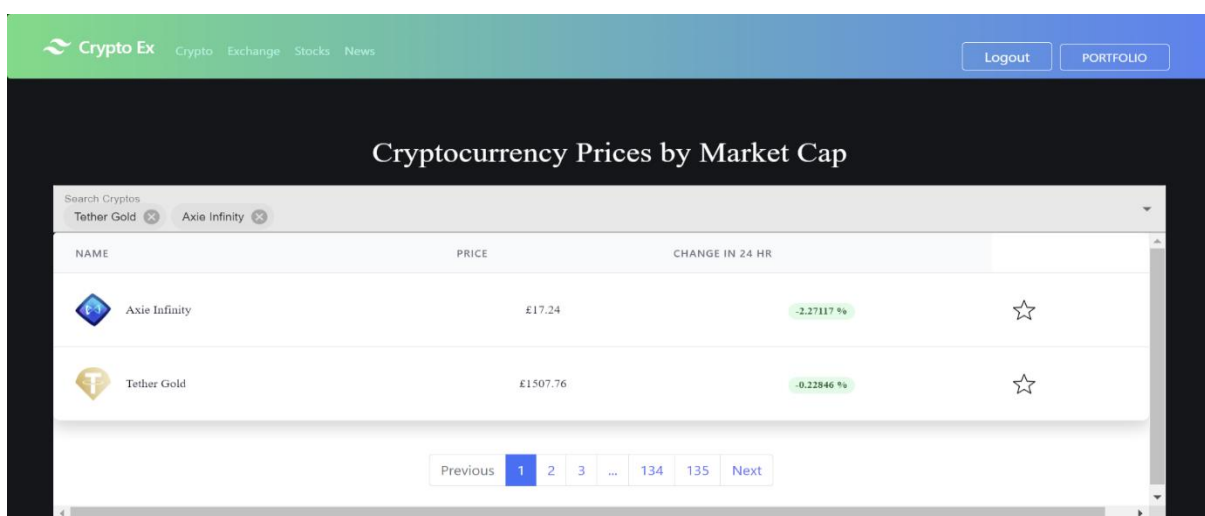
\*HomePage component side code

The favourites feature implementation is a tricky process to start with. Should start to work with the state of the portfolio component and handleSubmit() function of the HomePage component simultaneously. The concept of pagination is crucial. Pagination in simple terms is rearranging the selected data of various source pages into a readable new page. Here, the new page is the portfolio page consisting of favourites and source pages are the various crypto pages. Initially, the implementation starts with parsing the JSON data of pre-existing favourites in the local storage that is retrieved from an external API i.e., Coin Gecko through the token of the respective coin. After the completion of the initial loading, the portfolio component works whenever the user deals with the favourite star icon on the crypto page. The handleSubmit() function is present both in the portfolio component as well as HomePage component. As the favourites is an array, whenever the user clicks star button to add to the portfolio, there would be filter method where it checks for the presence of it in the pre-existing favourites array. If the selected coin doesn't exist in the array, it gets added to the favourites. Then the patch method is introduced to update the pre-existing favourites array on the backend. Patch method simply adds or deletes a single element of an array efficiently. But



before patching the new favourite coin with the favourites array, the JS coin object is converted to the string by using `JSON.stringify()`. The resultant favourites array that is updated on the web server is outputting in the portfolio page. The local storage of the browser where the credentials and user's inputs are saved temporarily updates its JSON data with respect to the web server's updated favourites array data. A method of the event interface called `e.stopPropagation()` where 'e' is the event is used to prevent the future expansion of the present event for the capturing and bubbling. This further helps in unselecting the favourite button if the user changes to dislike the coin. This looping between selecting and unselecting the favourite button results in the event bubbling where it runs through the handlers between the HomePage component and the Portfolio component retrospectively.

## 7.9 Search component



The search component deals with the hunting of a specific crypto by the user. The user can just search for the crypto to view they wish for. This component also helps in a quick comparison of two or more cryptos with their respective prices and change in percentage in 24 hours. There is also a delete selection on each coin in the search label which can be used to omit that specific coin out of the search list. The components like TextField and



Autocomplete that are obtained by Material UI (MUI) are used to create a filled search label as well as add functionality and design to the search label whenever the user uses it.

```
export default function SearchComponent(props) {
  const [open, setOpen] = React.useState(false);
  const [options, setOptions] = React.useState([]);
  const loading = open && options.length === 0;

  const onChangeHandle = async value => {
    if(value.length >=3 ) {
      const response = await axios.get(props.url+value);
      setOptions(response.data[props.dataString]);
    }
  };

  React.useEffect(() => {
    if (!open) {
      setOptions([]);
    }
  }, [open]);

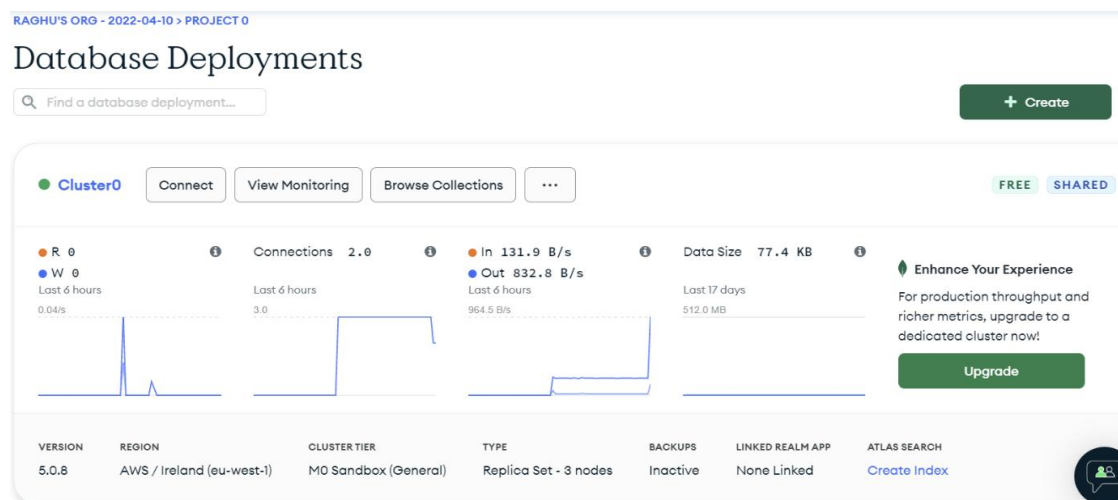
  return (
    <Autocomplete
      id="asynchronous"
      style={{ backgroundColor: "#fff" }}
      open={open}
      multiple={true}
      onOpen={() => {
        setOpen(true);
      }}
      onClose={() => {
        setOpen(false);
      }}
      onChange={onChangeHandle}
      getOptionSelected={option => option.name === value.name}
      getOptionLabel={option => option.name}
      options={options}
      loading={loading}
      renderInput={params => (
        <TextField
          {...params}
          label="Search Cryptos"
          variant="filled"
          onChange={onChangeHandle}
          // dont fire API if the user delete or not entered anything
          if (ev.target.value !== "" || ev.target.value !== null) {
            onChangeHandle(ev.target.value);
          }
        >
        <InputProps={
          ...params.InputProps,
          endAdornment: (
            <React.Fragment>
              {loading ? (
                <CircularProgress color="inherit" size={20} />
              ) : null}
              {params.InputProps.endAdornment}
            </React.Fragment>
          )
        }
      >
    </Autocomplete>
  );
}
```

This functional component is passed as props to the homepage component where it is present. The state of this component is dependent on two methods, namely setOpen() and setOptions(). The open variable emphasizes whether any coin is searched and selected as well as acts as the dependency for useEffect() hook where opening or closing of any particular crypto refreshes the whole component and updates its state. The Autocomplete component of MUI helps in searching a particular coin from the predefined set of crypto values. The options variable demonstrates about the list of the cryptos that are present in the suggestions. The TextField component of MUI provides the functional label for searching cryptos. The onChangeHandle() is an asynchronous function used to handle the selected the coins from the suggestion list and place them in the search label adjacent to previously selected coins. The CircularProgress is a MUI component used here to introduce the loading template that is adjacent to the search label. If the user isn't specific about the name of the coin they are looking for, they can just

move through the crypto pages that are present at the footer of the container and search the name to compare it with searched coin group.

## 8. Backend Development

The backend development for this project starts with the creation and deployment of the database. Here, the database we use is MongoDB which is a no-SQL database.



### 8.1 Data Schema

For any no-SQL database, the data schema or the data model is crucial in creating collections of data. In simple terms, a data model gives a blueprint of the data structure that we will be creating on the no-SQL database. The data object modelling tool for MongoDB called Mongoose is used to design the data schema for the asynchronous data. There are five data schemas that are being connected with the MongoDB for this project. Each user has their own set of data schema values ranging from name, email, password, date of creation and their favourites list. This format of defining the structure as well as the contents of the data results in creation of the collections array in the MongoDB that is internally connected with the cloud for the storage of user data.

```
const mongoose = require("mongoose");
const { Schema } = mongoose;
const UserSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  date: {
    type: Date,
    default: Date.now,
  },
  favourites: [{ type: String }],
});

const User = mongoose.model("user", UserSchema);
module.exports = User;
```

## 8.2 Middleware Setup

Before creating the routes for the backend to receive data, setting up the middleware is essential. Middleware is basically a piece of code that runs on the server in between receiving the request and sending the response by the server. For this project, middleware is being used for the authentication check of the user using the JWT(JSON Web Token). JWT is an open security standard for verifying the client and server before the exchange of data takes place. Here, '.env' file consists of the URL for MongoDB as well as the secret key for JWT. So, whenever a server gets a request, the middleware initializes the JWT to create a token to authorise the user's data on the server. If the token that is created is invalid or possess any error, the response from the server would be an error. This results in the verification of the user with their own JSON data on the server. If the verification of the user completes using JWT, the routing mechanism initializes to authorise the user according to their requested routes. The JWT provides an extra security feature in verification of the user with the server as it is a URL protected, closely encoded and cryptographic signed string. After the request is authorised by JWT, the server sends the required response to the user based on their selected routes.

```

var jwt = require("jsonwebtoken");
require("dotenv").config();

const JWT_SECRET = process.env.JWT_SECRET;

const fetchuser = (req, res, next) => {
  // Get the user from the jwt token and add id to req object
  const token = req.header("auth-token");
  if (!token) {
    res.status(401).send({ error: "Please authenticate using a valid token" });
  }
  try {
    const data = jwt.verify(token, JWT_SECRET);
    req.user = data.user;
    next();
  } catch (error) {
    res.status(401).send({ error: "Please authenticate using a valid token" });
  }
};

module.exports = fetchuser;

```

### 8.3 Backend Routing

Routing on the backend is essential in getting the accurate responses from the server. For this project, the backend routing consists of two main routes i.e., '/api/auth/createuser' and '/api/auth/login'.

#### 8.3.1 Route – 1 '/createuser'

```

router.post(
  "/createuser",
  [
    body("name", "Enter a valid name").isLength({ min: 3 }),
    body("email", "Enter a valid email").isEmail(),
    body("password", "Password must be atleast 5 characters").isLength({
      min: 5,
    }),
  ],
  async (req, res) => {
    // If there are errors, return Bad request and the errors
    const errors = validationResult(req);
    let success = false;
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    // Check whether the user with this email exists already
    try {
      let user = await User.findOne({ email: req.body.email });
      if (user) {
        return res
          .status(400)
          .json({ error: "Sorry a user with this email already exists" });
      }
      const salt = await bcrypt.genSalt(10);
      const secPass = await bcrypt.hash(req.body.password, salt);

      user = await User.create({
        name: req.body.name,
        password: secPass,
        email: req.body.email,
      });
    } catch (error) {
      console.log(error);
    }
  }
);

```

This route is the backend routing functionality for the Signup page component on the frontend. The body consists of three variables with respective labels having the minimum length constraints for both name and password respectively. As the data movement between the client and server is asynchronous, the errors for this route are demonstrated, either by the bad request to the server or by pre-existential user signup by checking their email. The `findOne()` method is used to check whether the email was used previously by any user. For the user password security and unreadability by third parties, a hashing function called Bcrypt is used. Salt is a hashing process which is in simple terms adding a unique and random string before storing the B-crypted user password. Bcrypt along with Salt makes the user password 128 bit encrypted and results in unpredictability. By the above code, the initial user created password undergoes a set of hashing on its body through Bcrypt and prefixing of 10 random characters to the hashed password through Salt.

### 8.3.2 Route-2 '/login'

```

router.post(
  "/login",
  [
    body("email", "Enter a valid email").isEmail(),
    body("password", "Password cannot be blank").exists(),
  ],
  async (req, res) => {
    // If there are errors, return Bad request and the errors
    const errors = validationResult(req);
    let success = false;
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { email, password } = req.body;
    try {
      let user = await User.findOne({ email });
      if (!user) {
        return res
          .status(400)
          .json({ error: "Please try to login with correct credentials" });
      }

      const passwordCompare = await bcrypt.compare(password, user.password);
      if (!passwordCompare) {
        return res
          .status(400)
          .json({ error: "Please try to login with correct credentials" });
      }

      const data = {
        user: {
          id: user.id,
        },
      };
      const authtoken = jwt.sign(data, JWT_SECRET);
      res.json({ success: true, authtoken, user });
    } catch (error) {
      console.error(error.message);
      res.status(500).send("Internal Server Error");
    }
  });

router.patch("/update", async (req, res) => {
  var _id = req.body._id;

  const user = await User.findById(_id);
  if (user) {
    user.favourites = req.body.favourites;

    const updateUser1 = await user.save();
    res.status(200).json(updateUser1);
  } else {
    res.status(404);
    throw new Error("User not found");
  }
});

```

This route is the backend routing functionality for the login page component on the frontend. As the user data is pre-existed on the database, `findOne()` method helps in getting the exact user with the identical username i.e., email. If the email doesn't exist in the database, it returns error. The input password field string is being compared with the b-crypted password that existed in the database. If both the passwords are not identical, it returns error. Each user has their unique id whenever they signup. After authenticating user with the login form, the user id acts as an authentication token to access the respective user data with signing the JWT. The response from the server results in login of the user securely. The patch method on the router enables



the server in adding or deleting a piece of data to an existed array. Here, it is used to handle the user's selection of their favourite coins. Whenever, the user selects or unselects the favourite button, the patch method routes the database on the server to follow the user inputs.

## 8.4 Server-Side Rendering

For this application, the server-side rendering is carried out for fetching the data from third party API and sending it to the client-side Stocks component where the output is displayed according to the fetched data by the server. In simple terms, server-side rendering is implemented to render the pages on the server rather than traditional browsers. Here, this approach helps in more reliability on server, faster updates, and better loading times. The server-side rendered code here is divided into three routes for the stock data. These routes are '/search', '/details' and '/timeseries'. So, three of these routes are simultaneously rendered on the server whenever the user accesses the stock page component on the frontend.

```
const express = require("express");
const request = require("request");
const router = express.Router();
const API_KEY = "01V2YH5II47S20BQ";

// Route:-1 search stock
router.get(
  "/search",
  async (req, res) => {
    const query = req.query.query;
    // If there are errors, return Bad request and the errors
    var url = `https://www.alphavantage.co/query?function=SYMBOL_SEARCH&keywords=${query}&apikey=${API_KEY}`;

    await request.get({
      url: url,
      json: true,
      headers: {'User-Agent': 'request'}
    }, (err, response, data) => {
      if (err) {
        res.send(err);
      } else if (response.statusCode !== 200) {
        res.send(response);
      } else {
        // data is successfully parsed as a JSON object:
        res.json(data);
      }
    });
  }
);
```

The code above depicts the use of Express JS to build the backend for the stocks page along with its router capabilities. The third-party API that was used here was Alpha Vantage. Its key had been declared at the start to use it for multiple routes for this page. The request comes from the frontend to search for the stocks where the query arises to check all the search symbols that are present in Alpha Vantage. So, the promised request would be received and response list in the form of search data would be sent back to the frontend.

```
router.get(
  "/details",
  async (req, res) => {
    const symbol = req.query.symbol;
    // If there are errors, return Bad request and the errors
    var url = `https://www.alphavantage.co/query?function=OVERVIEW&symbol=${symbol}&apikey=${API_KEY}`

    await request.get({
      url: url,
      json: true,
      headers: {'User-Agent': 'request'}
    }, (err, response, data) => {
      if (err) {
        res.send(err);
      } else if (response.statusCode !== 200) {
        res.send(response);
      } else {
        // data is successfully parsed as a JSON object:
        res.json(data);
      }
    });
  }
);
```

Whenever the user selects a stock from the search list on the frontend, that particular stock's details would be accessed by the backend from Alpha Vantage and sent it back to the user. The above code depicts the flow of data for a particular stock between the frontend and backend. Initially, front end user requests a particular stock through its symbol where the backend, after receiving the request fetches the data by accessing the stock symbol with its API key. The server then sends the response which is in JSON format to the client and outputs that selected stocks details on the left part of the stocks page.



```
router.get(
  "/timeseries",
  async (req, res) => {
    const granularity = req.query.granularity;
    const symbol = req.query.symbol;
    // If there are errors, return Bad request and the errors
    var url = `https://www.alphavantage.co/query?function=${granularity}&symbol=${symbol}&apikey=${API_KEY}`;
    if(granularity==="TIME_SERIES_INTRADAY") {
      url+='%&interval=5min'
    }

    await request.get({
      url: url,
      json: true,
      headers: {'User-Agent': 'request'}
    }, (err, response, data) => {
      if (err) {
        res.send(err);
      } else if (response.statusCode !== 200) {
        res.send(response);
      } else {
        // data is successfully parsed as a JSON object:
        res.json(data);
      }
    });
  }
);
```

Along with the stock details, there is a requirement for the component to show the graphical representation of the stock prices from a particular timeline. So, '/timeseries' path provides the rendering of the data required for the frontend component to output in the graphical interface. Here, the server renders various time series of the stock prices divided by granularity. Whenever the user requests the time series data for that particular searched stock, the server fetches the granular timeline price day based on the symbol name from Alpha Vantage and sends the response to the browser to output it on the left side of the stocks page in the form of graphical data.

### 8.5 Connecting Front-end with Back-end

The flow of data between the user inputs on the front-end and data instructive cycle on the back-end is crucial in derailing the data flow from the unnecessary back-end APIs to the exact path. There are three instances in the project where the data flow takes place between the front-end and the back-end. The three instances are whenever the user tries login, signup and manage their favourites. Each instance has their unique backend URL to access the

specific dataset that was created or going to be created for that particular instance. Based on the usage of the instance, either GET, POST, PUT, PATCH or DELETE is used as the HTTP method to start the data flow. The content-type representational header is used to detail about the data format of the user input data. As the server-side data is in the string format, the browser's JSON data should be converted to string before sending the user input to the server. So, a method called `JSON.stringify` is used to convert the JSON user input to the string format that can be understood by the server.

```
const handleSubmit = async (e) => {
  e.preventDefault();
  const response = await fetch("http://localhost:5000/api/auth/login", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      email: credentials.email,
      password: credentials.password,
    }),
  });
  const json = await response.json();
  console.log(json);
  if (json.success) {
    props.showAlert("login successfully", "success");
    //save the auth token and redirect back to
    localStorage.setItem("token", json.authtoken);
    navigate("/");
  } else {
    props.showAlert("invalid credentials", "danger");
  }
}
```

The first instance of transfer of data between front-end and back-end is whenever the user tries to login and access the application. Filling up the form and clicking the login button instantiates the connection between the browser and the server. Initially, the user input data in the form is converted to string

for the server to understand. The post method gets the converted string to the specific backend hosting URL namely `‘/auth/login’` for this login component. The authentication of the user takes place by comparing the posted form details with the pre-existential user data on the database. Then, the server sends a JWT (JSON Web Token) to authorise that specific user whenever they use a feature of that application, and it is temporarily stored in the local storage of the browser. This whole initial login process is being run as a promise that awaits the response from the backend after posting it.

```
const handleSubmit = async (e) => {
  e.preventDefault();
  const response = await fetch("http://localhost:5000/api/auth/createuser", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      name: credentials.name,
      email: credentials.email,
      password: credentials.password,
      cpassword: credentials.cpassword,
    }),
  });
  const json = await response.json();
  console.log(json);
  if (json.success) {
    props.showAlert("account created successfully", "success");
    //save the auth token and redirect back to
    localStorage.setItem("token", json.authtoken);
    navigate("/");
  } else {
    props.showAlert("email already exist", "danger");
  }
};
```

The second instance of data transfer between the front-end and the back-end in the application is for the user signup feature. The application can only be accessed by the user having a login account. So, signup is the first step to create the login credentials. The flow of data for this instance is similar to

that of the login feature as the methods used to work on the backend API are similar i.e., POST. But the main difference is when the signed-up user gets navigated to the login page instead of the homepage as it is for the login feature. Likewise in the login feature, it creates an alert after signup and sets JWT with the server but only authorises it whenever the user login into the navigated page after the signup. There is also an additional conditional here with the verification of the pre-existing user details on the server and proceeds further if there is no danger alert.

```
const response = await fetch("http://localhost:5000/api/auth/update", {
  method: "PATCH",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({
    _id: localStorage.getItem("_id"),
    favourites: fav,
  }),
});
const json = await response.json();
setFavourites(json.favourites);
console.log(json);
localStorage.setItem("favourites", JSON.stringify(json?.favourites));
};
```

The third and the last instance where the data flow takes place between the frontend and the backend is whenever the user tries to add or remove the favourites coin with toggling between the star icon that is placed for each coin. Whenever the user toggles, the front-end connect to the back-end API with a path '/auth/update'. The data fetching starts from the path with the method PATCH which basically modifies the contents of the favourites array by adding or deleting a coin by the user's toggling mechanism. Each user's distinct id manages the array list of their favourites. Here, the method setFavourites() is called whenever there is change in the state of the favourites array value and stores the new list temporarily in the local storage before sending and updating the array that is present in the database.

## 8.6 Development Packages used

Node JS supports extensive libraries of packages ranging from multi used to single used ones to reduce the development time. The list of Node JS packages that are used for the development of this application are mentioned below.

- i. [nodemon](#) – The basic and most extensively used tool for Node JS based applications for the automatic restart of the application whenever there is a change in the contents of the files in the directory.
- ii. [bcrypt](#) – The library for the implementation of the hashing in user passwords.
- iii. [cors](#) – The Cross-Origin Resource Sharing (CORS) is used for the restriction of the cross-originated HTTP requests with other servers resulting in the browser security. It also specifies the domains that are accessing the resources.
- iv. [bcryptjs](#) – The optimization of JavaScript possessing zero dependencies through bcrypt.
- v. [dotenv](#) – It helps in storing and accessing the configuration files for the server in an environment away from the main code but can be directly accessed anywhere.
- vi. [express](#) – The open-source library for Node JS to create and manage the handlers for numerous HTTP requests that can be accessed at different routes.
- vii. [express-validator](#) – It is an Express JS middleware library that is used for the data validation on the server-side.
- viii. [mongoose](#) – A tool for MongoDB used for the object modelling of the asynchronous data containing both promises and call-backs.
- ix. [jsonwebtoken\(JWT\)](#) – An open standard for providing security to the sharing data between the client and the server. In simple terms, JWT is the authorisation technique to authorise the client by the server whenever there is a request by the client to the server. JWT is the replacement of age-old method of authorising the client by the server with session ids that are stored as cookies in the client's browser. JWT eliminates the use of storing the authorised data on the server's database. Initially, the server creates JWT for the client with a secret key

and sends it to the client. Whenever there is a request by the client with the server, the server verifies the received JWT from the client with identical encoding and serialization that was previously signed by the server, thus authorising the particular user with their unique JWT every time they request something from the server.

## 9. Epilogue

### 9.1 Testing

The evaluation for any web application depends on testing. The evaluation plan ranges from various checklists ranging from design to functionality. The user-interface and functionalities are the important aspects of this web application testing. The methods for the evaluation of this application are broadly divided into two categories. They are functional testing and non-functional testing.

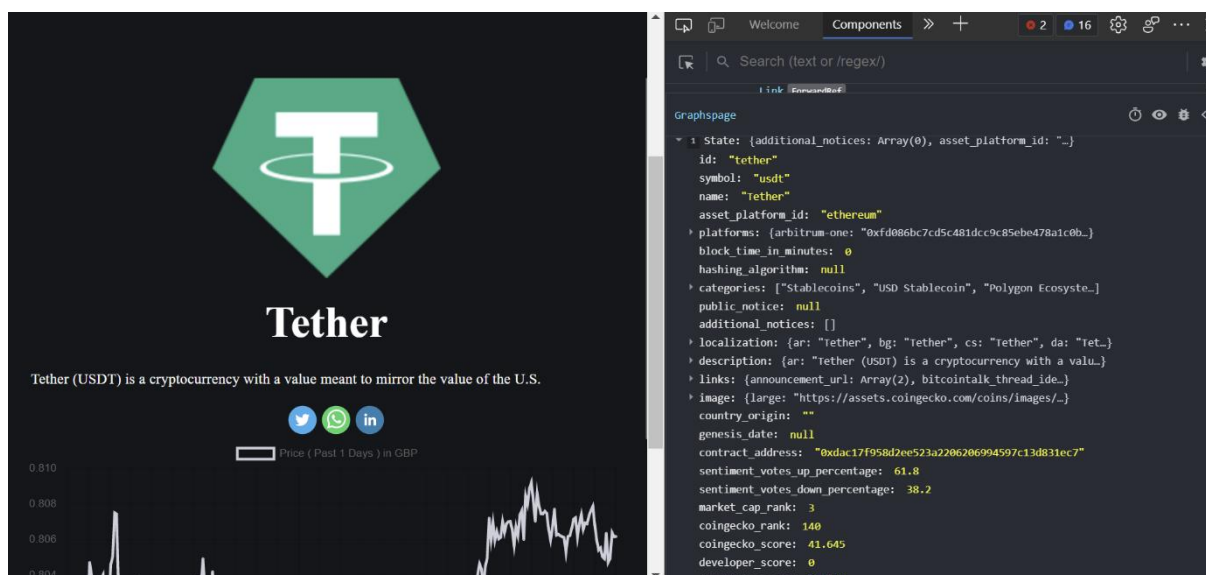
#### 9.1.1 Functional Testing

This categorized part of testing deals with the validation of the application against the functional requirements or specifications. Here, its main purpose is testing each functional component of the application through providing respective input to verify the output against their intended initial functional requirement. This manual type of black box testing is irrespective of this application's source code and tests only the user-interface, flow of data from third party APIs to application, database functionality, communication between client and server, and the security of the application.

Initially, the testing had started with the user-interface. Here, each page is thoroughly scrutinized for the design moulds. Each page possesses unique way of outputting the data. The outline design of all the pages is carried out by Material UI (MUI). The working of dark theme only on the specified pages is tested. For example, the news component doesn't use a dark background. The homepage and exchanges page output the containers with tabular rows and columns. So, those pages are tested whether they output the intended form of the user-interface with perfect sync of the rows and columns with the data. The search component for the homepage as well as stocks page is thoroughly tested to meet its requirement for outputting the loading template while

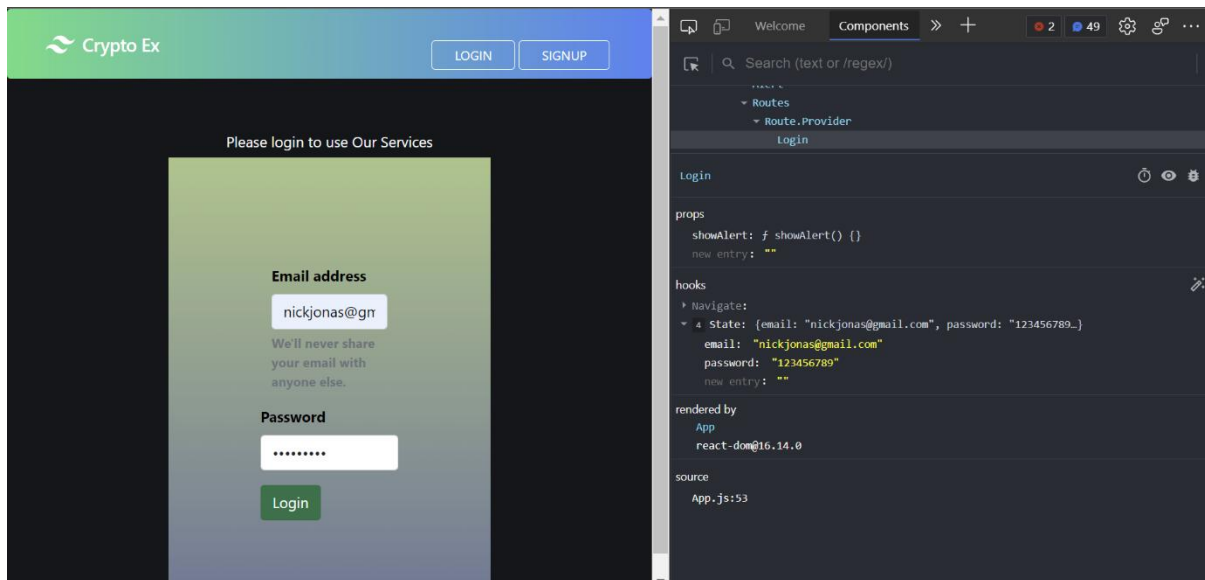


searching along the intended output list of suggestions right down the search label. When opening a particular crypto or stock component, the graphs page should output the naming and specification of the intended data to the left side of the component along with correct placing of the social media icon buttons for the user to share that particular coin with others and the output of graphical data to the other side of the component, thus meeting the intended requirement for the user to toggle down on the responsive timeline objects. The portfolio page is thoroughly scrutinized for providing the intended starred button for each coin along with exact output of the intended tabular form of data with rows and columns as well as exact placing of the social media icon buttons for each starred coin for the user to share with others. For the entire application, running of the loading templates was thoroughly trailed out.



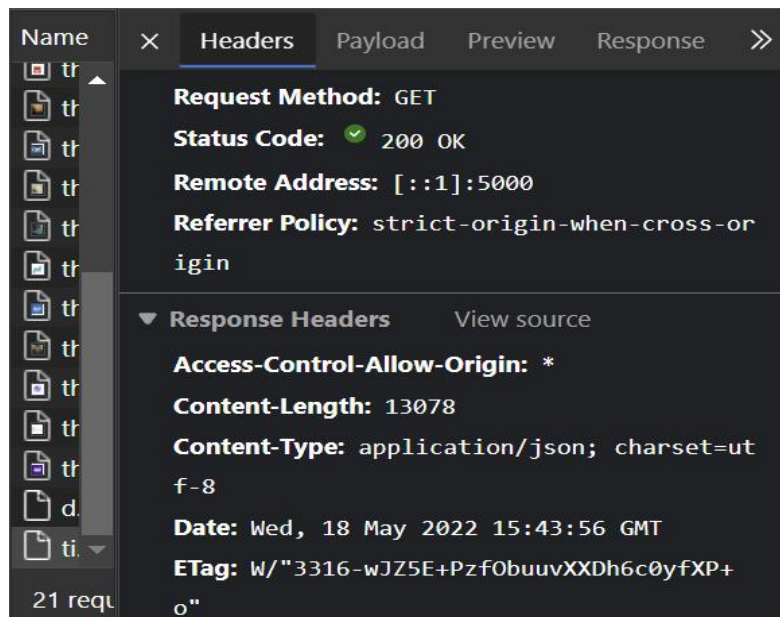
By downloading and adding the browser add-on called React Developer Tools, we can practically access and test each individual component. The above screenshot depicts the pathname as well as contents of a specific crypto called 'Tether' from a third-party API. This way of testing each part of output where the data comes from the third-party APIs results in zero error of fetching the real time data. Initially, after signing up, the database had been verified multiple times whether it updates the new user data to its existing user collection data. As the user's login and favourites are stored in the

database, the favourites array is tested multiple times by adding to it and removing from it and verifying that on the Mongo database.



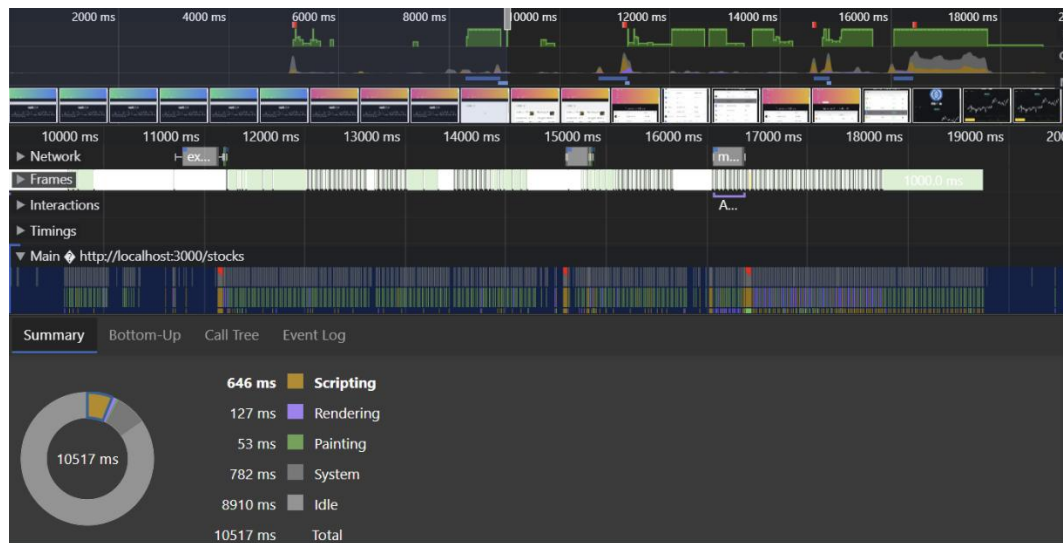
The input of the form validation is tested by using the components tool of React Developer Tools. Whenever, the user input data into form, the browser should be able to grasp that data by change in the state of the application every time user modifies. This is crucial in the form validations as well as for managing the input in search component. For the user security of this application, JWT (JSON Web Token) plays an important role. The user login password would be hashed with Bcrypt, and Salt will be introduced on it to make it unpredictable before saving that on the database. JWT helps in authorising the logged in user every time they navigate between multiple routes in the application. The network tab of the developer tools is crucial in testing the flow of data between the frontend and the backend as well as functioning of the JWT. The screenshot below depicts the GET request to the client from the server. Here, the requested data is for the user's portfolio. ETag is the JWT key that authorises the logged in user's identification with that particular user's data present on the database. Here, the requested portfolio data had been sent by the server to the user's browser.





### 9.1.2 Non-Functional Testing

For this application, the main emphasis of the non-functional testing is applied to the performance. The network loading times, the rate of flow of data between frontend and backend, the data retrieval speeds from the third-party APIs and overall network latency are thoroughly tested and implemented successfully. The network tab on the browser's developer tools clearly depicts the network performance, latency as well as data packets lost while transmission. It also can block cookies for the user's identity leak to third party APIs. There is also a performance tab on the developer tools which can record the animation speeds, network stability, framerate, data rendering speeds and network latency in a productive environment.



## 9.2 Challenges faced

There are various challenges that the developer had faced during this web application development. Some of them are as follows.

- The developer had little to no experience in developing the backend systems at the start of the project. To overcome that, I had extensively worked on various backend documentations ranging from Node JS, Express JS, and MongoDB as well as practicing to work with various backend APIs.
- The developer's knowledge on financial markets as well as block-chain related cryptos are only restricted to the theoretical part before the start of the project. By referring various published papers and journals that are mentioned in section 3.1 as well as online video tools to gather deep information on them helped me to add more functionalities to the project.
- The combination of Tailwind CSS, Material UI (MUI) and Chart JS for the frontend development was done by the developer for the first time. So, during the graphs page designing, various ideas were developed and tested but took quite a long to get to the present design.
- As it is a full stack web application, the rectification of internal errors during the transfer of data between frontend and backend in the initial stages of the development had been quite challenging.

### 9.3 Future Development

The developer believes that this web application of Crypto Investment Assistant has a great prospect if it is enhanced further to the level where the user can perform all the block-chain related tasks in one application. It may be quite a long way to reach there but the steps that are mentioned below can make it happen.

- i. Adding more crypto related functionalities like real-time animations of price change, selling, and buying the cryptos by establishing a crypto exchange, providing more extensive and elaborative timelines for the prices, adding more third-party APIs to gather more news to the news component and also implementing smart contracts.
- ii. Providing the user with two-step authentication for more security. Implementing to gather more information on the user's signup page and also verification of the user's email during signup.
- iii. Improving the stock recommendations for the user by their previously searched data.

## 10. References

1. Monika Mehra, Manish Kumar, Anjali Maurya, Charu Sharma & Shanu 's MERN Stack Web Development. Available at: <https://www.annalsofrscb.ro/index.php/journal/article/view/7719/5721>
2. Samikshya Aryal 's MERN Stack with modern web practices. Available at: [https://www.theseus.fi/bitstream/handle/10024/338237/Samikshya\\_Aryal\\_Final\\_Thesis.pdf?sequence=2](https://www.theseus.fi/bitstream/handle/10024/338237/Samikshya_Aryal_Final_Thesis.pdf?sequence=2)
3. Joel Vainikka 's Full-stack web development using Django REST framework and React. Available at: [https://www.theseus.fi/bitstream/handle/10024/146578/joel\\_vainikka.pdf?sequence=1](https://www.theseus.fi/bitstream/handle/10024/146578/joel_vainikka.pdf?sequence=1)
4. Mozilla Developer Resources. Available at: [MDN Web Docs \(mozilla.org\)](https://developer.mozilla.org/)
5. Node Package Manager library. Available at: <https://www.npmjs.com/>

6. Node JS Documentation for working on backend server environment.  
Available at: <https://nodejs.org/en/docs/>
7. Express JS, a backend framework's working documentation. Available at:  
<https://expressjs.com/en/5x/api.html>
8. React JS, a frontend framework's working documentation. Available at:  
<https://reactjs.org/>
9. Tailwind CSS, a utility first CSS framework's running documentation.  
Available at: <https://tailwindcss.com/docs/installation>
10. MUI's documentation, React UI library tool for designing frontend components. Available at: <https://mui.com/>
11. Online Tool to draw and organise initial project work consisting of Use-case and Architecture diagrams. Created at:  
<https://www.lucidchart.com/pages/>