Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

GlobeTrek

Project Engineering

Year 4

Parthib Barua

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Parthib Barua_____

Acknowledgements

I would like to express my sincere gratitude to my project supervisor, **Ben Kinsella**, for his invaluable assistance, support and guidance throughout the course of my final year project. His insights and expertise greatly contributed to successful completion of this work.

I would also like to extend my thanks to the Project Engineering coordinators **Paul Lennon**, **Michelle Lynch**, **Niall O'Keeffe**, **Brian O'Shea** and **Pat Hurney**.

Table of Contents

1 Summary 6
2 Poster 7
3 Introduction 8
4 Tools and Technologies 10
4.1 React 10
4.2 Node.js 10
4.3 Express 10
4.4 Mongoose 11
4.5 MongoDB 11
4.6 Amazon Elastic Compute Cloud 11
4.7 JSON Web Token (JWT) Tokens 11
4.8 React Router 11
4.9 AXIOS 12
4.10 Bcrypt 12
4.11 Google Map API 13
4.12 Flight API 13
4.13 Currency Exchange Rate API 13
4.14 Tourist Places 13
4.15 Weather API 13
5 Project Architecture 14
6 Project Plan 15
7 Project Code 17
7.1 Frontend 17

7.2 Backend 32

8 Ethics 42

9 Conclusion 43

11 References 47

1 Summary

For my final year project, I wanted to develop something that could address some real-world issues, even if most of them have already been resolved. So, I took inspiration from a hotel booking website, Booking.com and tried to create my own website that would facilitate hotel booking and management for travellers.

The scope of the project includes the development of both front-end and back-end components using the MERN stack (MongoDB, Express, React, and Node.js). Key functionalities include user registration and login, hotel search based on the cities and booking, hotel reviews, and personalized user dashboards, and secure authentication using JWT tokens and password hashing. It also integrates several third-party APIs, such as Google maps, Weather, Flight booking and status, Currency Exchange rate, and Tourist Spots, offering a comprehensive experience for travellers. Through this project, I accomplished the successful deployment of a live platform hosted on Amazon EC2, backed by MongoDB Atlas for data storage.

In addition to its core technical objectives, the project emphasizes planning and documentation. Tools like JIRA, Figma, OneNote, and GitHub were used for task planning, management, research, design, storing codes and resources.

This report details all the stages of the GlobeTrek like planning, designing, development, and testing platform. It also reflects on the challenges faced, the lessons learned, and the potential for future enhancements.

2 Poster

Here is my poster:

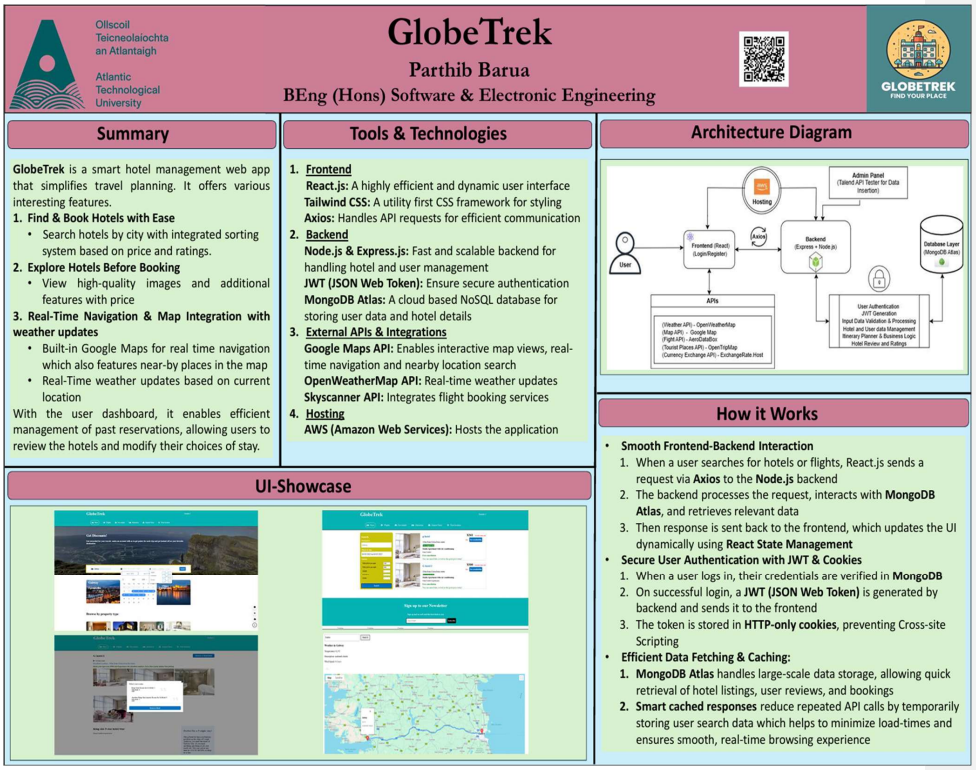


Figure 2.1. Poster

3 Introduction

My intention behind this project was to help travellers with a simple application that would allow them to manage their hotel booking process easily. By bringing everything under one roof, like Maps, Weather, a currency converter, Flight, and an external Taxi Booking Link for the tourists, not only simplifies the planning process but also enhances the overall travel experience. Furthermore, this project provided an opportunity to create a real-world, functional application while strengthening my skills in software development, project management, and problem-solving.

GlobeTrek is an online travel platform that helps travelers from all over the world to manage their hotel reservations. Additionally, it offers built-in APIs to help users. If users already have an account, they can log in; if not, they can register using their username, email address, and password. Hotel reservations can only be made by those who have accounts and are logged in. Users can sort the hotels based on the price. Also, this website lets customers read hotel ratings and reviews from prior visitors, which are managed using the Bayesian Review system. Additionally, the customer can browse through the hotel's photos after selecting any individual hotel. Each user has their own Profile Dashboard where they see their previous and upcoming bookings, and they can rate their stay with some comments.

This website also hosts some in-built third-party API's, which allow the user to get the opportunity to see their live locations using Maps integrated with weather updates. A system that lets users book flights and check flight statuses (departure and expected flights) based on the airport they have chosen. It has a Currency Exchange API that lets users see the most recent exchange rate. Finally, an API is hosted to check the nearby tourist places. Users can search for the nearby attractions and can select the page; upon selecting it, navigates the user to that location through an in-built map API.

The MERN stack was used to create this website. This project's development and hosting also required the use of technologies like React Navigation, React Router DOM, React Hooks, useFetch, Context, Mongoose, and Amazon EC2. The front-end uses React, Navigation, Hooks,

and useFetch, while the back-end uses Node.js, Express, and Mongoose and is hosted on an EC2 t3.medium instance. All the data that is hosted by MongoDB Atlas is stored in MongoDB. To manage and plan the project efficiently, I utilized JIRA, along with OneNote. JIRA helped me to plan and track the progress from the research to the development phase, while I used OneNote to store all my research and project notes by logging all the work on a weekly basis. Additionally, I first drew my project prototypes using Figma for the UI design, and then I utilized React to build those UIs.

Through this project, I developed my skills in project planning, management, and execution. I learnt how to bring an imagination to the life by proper planning, designing, and building. I improved my software skills through working with different technologies and built-in libraries. I used this project as an opportunity to learn Git/GitHub properly by storing all the codes in the GitHub. I managed my code base in GitHub by frequent commits, pushes and pulls. Overall, this project journey has helped to enhance my understanding of the website building process and problem-solving techniques.

4 Tools and Technologies

Selecting tools to build a full-stack website was very important. I was not familiar with MERN stack technologies. But it was essential to choose the right tools and technologies for implementing all the desired features.

4.1 React

React is a JavaScript library developed by Facebook for building user interfaces, particularly single-page applications. It is also referred to as a front-end JavaScript library. It allows developers to create reusable and interactive UI components. Every React component has a lifecycle of its own. Some key principles of React includes Component-based architecture, JSX syntax and Virtual DOM which make React as widely used web development tools. React operates by creating an in-memory virtual DOM rather than directly manipulating the browser's DOM [9].

4.2 Node.js

Node.js is an open-source and cross platform JavaScript runtime environment. Node.js is very performant and runs in a single process, without creating a new thread for every request. Node.js is not dependent on any operating system software. It can work on Linux, macOS, or Windows. Also, it allows us to run JavaScript code outside the browser, making it ideal for building scalable server-side and networking applications. Moreover, it enables the use of JavaScript for both frontend and backend development [12].

4.3 Express

Express is a minimal and powerful framework for building web applications and API's in NodeJS. When Integrated with MERN stack, Express handles server-side routing and provides a robust foundation for handling HTTP requests and responses. Routing, Middleware, Error Handling are some of the important features of Express. With these features, Express facilitates a vibrant ecosystem for easy integrations [2].

4.4 Mongoose

Mongoose is an Object Data Modelling (ODM) library for MongoDB and Node.js [34]. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB [3].

4.5 MongoDB

MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

4.6 Amazon Elastic Compute Cloud

Amazon Elastic Compute instance is a virtual server in the AWS cloud that offers scalable computing capacity. It allows us to launch and manage our virtual machines, known as instances. Instead of buying and managing own servers, EC2 gives us a virtual machine, where we can run websites, or even big data tasks. EC2 offers security, reliability, high performance, and cost-effective infrastructure to meet demanding needs [4].

4.7 JSON Web Token (JWT) Tokens

JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret or a public/private key pair. JWT are useful for some of its features like Authorization, Information Exchange and Web Token structure (header, payload, signature) [7].

4.8 React Router

React provides a library called React Router that simplifies the process of adding, routing and navigation to the applications. React router helps to define routes and components associated with those routes, making it easy to display the appropriate content based on the current URL. React Navigation is a JavaScript-based library that provides navigation

functionality using which developers can easily implement navigation in their web applications [28].

4.9 AXIOS

Axios is a popular JavaScript library used to make HTTP requests from the browser (React frontend) or Node.js backend. It helps React apps communicate easily with backend servers by sending or receiving data using simple promise-based syntax (async/await). Axios handles errors well, supports request customization (like adding authentication tokens), and simplifies connecting frontends to APIs. It's often preferred over the native fetch() due to its cleaner, more powerful features [30][32].

4.10 Bcrypt

bcrypt is a password-hashing library that securely encrypts user passwords before storing them, making it hard for attackers to reverse-engineer even if the database is compromised. It adds a random "salt" to each password and uses multiple rounds of hashing, making brute-force attacks much slower. In a Node.js backend, bcrypt is commonly used during user registration (to hash passwords) and login (to compare entered passwords with stored hashes). Its adjustable "cost factor" controls how computationally intensive the hashing is for extra security [35].

4.11 Google Map API

Google Map API is a JavaScript API which is a client-side web API that lets us create custom maps to show locations anywhere in the world. Mainly, it fetches the live location through the user's browser. It asks for the permission from the user before opening. If user allows, it fetches location of the browser or through the network connection and return the co-ordinates. Based on the co-ordinates, it reveals the live location with a marker in the map.

4.12 Flight API

AeroDataBox API is used to fetch flight related details. It is a cost-effective aviation and flights API which offers a wide range of data, including flight status, schedules, aircraft information, airport details, statistics, historical data, and more.

4.13 Currency Exchange Rate API

Exchangerates.host API is an API service for real-time foreign exchange. This API uses JSON format for the information delivery which ensures maximum usability and fast conversion.

4.14 Tourist Places

OpenTripMap API is allows to fetch worldwide points of interest database for travel and entertainments. It uses HTTP requests to get object data from **OpenTripMap** database. **OpenTripMap** based on cooperative processing of different open data sources like Wikipedia, Ministry of Culture, OpenStreetMap.

4.15 Weather API

OpenWeatherMap API provides current weather data, forecasts and historical data for any geographic location.

5 Project Architecture

Architecture Diagram

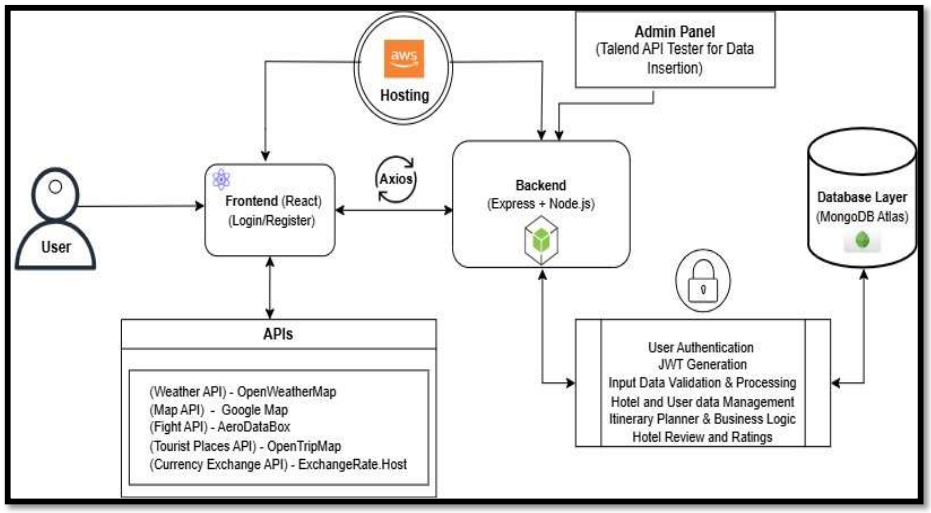


Figure 5-1. Architecture Diagram

6 Project Plan

To plan and track my project, I used JIRA. I created timelines for my research, design, and experiments. I updated it on weekly basis managing sprints and strictly followed the timeline. When I started using it during the 1st semester, I barely realized how to use JIRA efficiently but by the 2nd semester, I learnt to manage JIRA effectively and utilized it to manage the plan and track the progress of my project.

Every time I used JIRA; I created an epic in the timeline and a sprint in the backlog. Then I created tasks in the backlog for that sprint and connected those tasks with the epic associated with it. Then defined the start and end date with some meaningful goal for the sprint and started it.

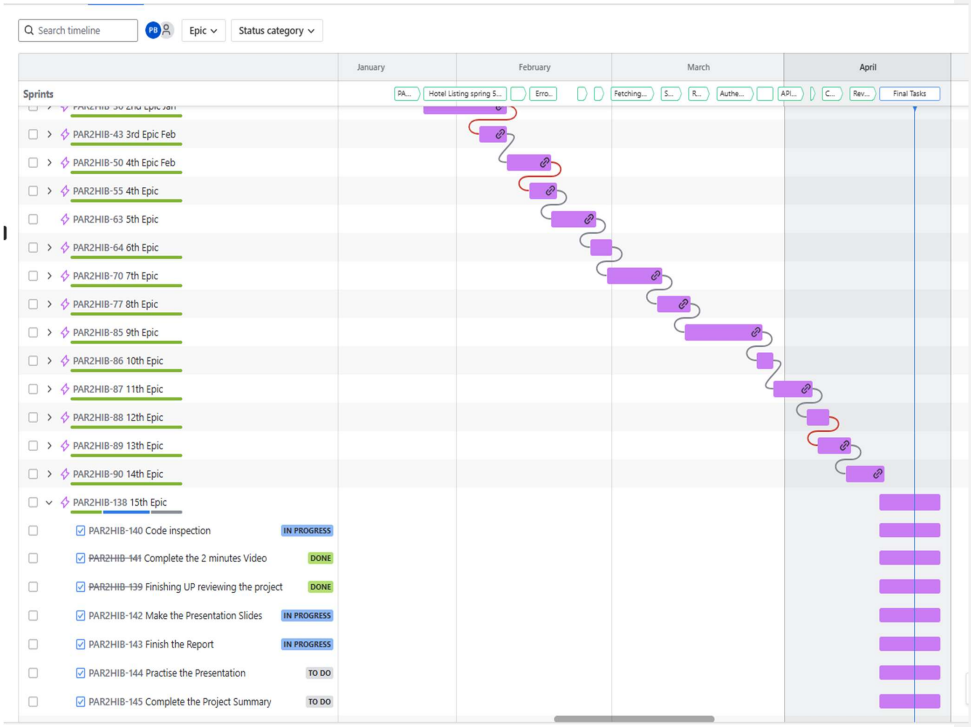


Figure 6.1. JIRA

I have also used Figma to plan and design my UI. This tool is helpful and customizable. Before building the UI, I visualized the idea and drew the idea here. Although I took inspiration from various hotel booking websites, this tool helped to put everything together.

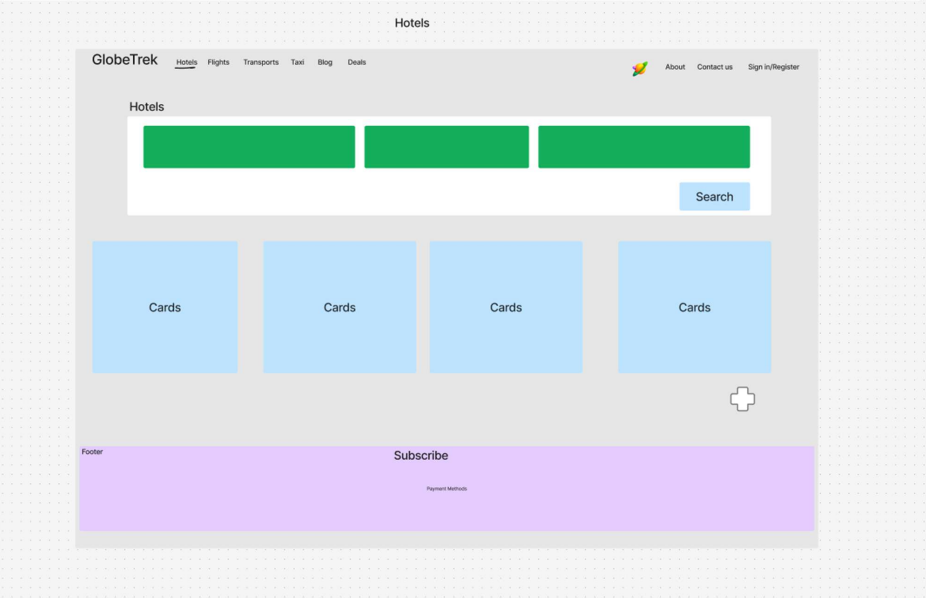


Figure 6.2. FIGMA

Finally, I used OneNote to store all my research and project notes. I have all the details about my project logged in there. Also, the teamwork section is available in OneNote as proof of the weekly team activity I have done throughout the year.

7 Project Code

I'll use the code in this part to highlight some of my project's key features and capabilities. In essence, I'll be outlining key front-end and back-end codes.

7.1 Frontend

These are my front-end codes architecture. I created separate folders for different components of my front-end UI. Elements from different folders are responsible for different tasks.

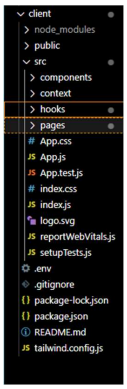


Figure 7.1.1. Frontend Code Architecture

In App.js, I declared routes for all the pages and components that my front-end features.

```
function App() {
  return (
    <BrowserRouter>
    <Routes>
      <Route path = "/" element = {<Home />} />
      <Route path = "/hotels" element = {<List />} />
      <Route path = "/hotels/:id" element = {<Hotel />} />
      <Route path="/email" element = {<MailList />} />
      <Route path="/map" element = {<Map />} />
      <Route path="/contact" element = {<Contact />} />
      <Route path="/weather" element = {<Weather />} />
      <Route path="/login" element={<Login/>} />
      <Route path="/register" element={<Register/>} />
      <Route path="/flight" element={<Flight/>} />
      <Route path="/profile" element={<ProfileDashboard />} />
      <Route path="/touristPlaces" element={<TouristPlaces />} />
      <Route path="/currency" element={<Currency />} />
      <Route path="/hotelDetails/:id" element={<HotelDetails />} />
      <Route path="/hotels-by-type" element={<HotelType />} />
    </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Figure 7.1.2. App.js Routes

In Home.jsx, I have the home page setup. This jsx structure is composed of various custom components like Navbar, Header, Footer, Featured, PropertyList, FeaturedProperties, HumbergerButton. All these customized components have their own functionality.

```jsx
export const Home = () => {
  return (
    <div>
      <Navbar />
      <Header />
      <div className="homeContainer">
        <Featured />

        <h2 className="homeTitle">Browse by property type</h2>
        <PropertyList />
        <h2 className="homeTitle">Peoples Favorite</h2>
        <FeaturedProperties />
        <HamburgerButton />
        <Subscribe />
        <Footer />
      </div>
    </div>
  )
}

export default Home;
```

Figure 7.1.2. App.js Routes

This implementation features my navbar for the whole frontend. It hosts the title of the website along with the 'Login' and 'Register' buttons. If the user is logged in, it will only show the 'Username'. If the 'Username' buttons gets clicked, a drop-down menu will pop up holding two more buttons called 'Profile dashboard' and 'Logout'.

Figure 7.1.3. Navbar Functionalities 1

There are few more buttons in the Navbar. User will be redirected to a new page if any button gets clicked. Each buttons have their own functionalities. This ‘ <span onClick={() => navigate(“/touristPlaces”)}>Attraction’ navigates through the ‘touristPlaces’ page. Here ‘useNavigate’ from ‘React Router DOM’ library is used to navigate between pages. Also, ‘FontAwesome’ icons are used which is imported from ‘react-fontawesome’ library [17].

```
<div className="navItems">
  <div className="navbarList">
    <div className="navbarListItem">
      <FontAwesomeIcon icon={faBed} />
      <span onClick={() => navigate("/")}>Stays</span>
    </div>

    <div className="navbarListItem">
      <FontAwesomeIcon icon={faPlane} />
      <span onClick={() => navigate("/flight")}>Flights</span>
    </div>
    <div className="navbarListItem">
      <FontAwesomeIcon icon={faTaxi} />
      <a
        href="https://www.free-now.com/ie/"
        target="_blank"
        rel="noopener noreferrer"

      >
        Taxi
      </a>
    </div>
    <div className="navbarListItem">
      <FontAwesomeIcon icon={faBed} />
      <span onClick={() => navigate("/touristPlaces")}>Attraction</span>
    </div>
    <div className="navbarListItem">
      <FontAwesomeIcon icon={faCoins} />
      <span onClick={() => navigate("/currency")}>Currency Exchange</span>
    </div>
    <div className="navbarListItem">
      <FontAwesomeIcon icon={faLocation} />
      <span>Your location</span>
    </div>
  </div>
```

Figure 7.1.3. Navbar Functionalities 2

This figure shows an example how the react ‘useState’ and a custom ‘useFetch’ hook is initialized. It sets up the states to handle location, state, and data fetching in a React component. It uses ‘useLocation()’ to get the current URL and extract an ID. ‘useFetch’ hook fetches hotel data with this custom hook. Also, a ‘dayDifference’ function is defined to calculate the number of days between two dates. The dayDifference function calculates the number of full days between two given dates. It first gets the time difference in milliseconds using the getTime() method for both dates and takes the absolute value to avoid negative results. Then, it divides that difference by the number of milliseconds in a day to convert it to days. Finally, it uses Math.ceil() to round up, ensuring that any partial day counts as a full one. This is particularly useful in booking systems where even a few hours can count as a full day for pricing or availability.

```
const location = useLocation();
// console.log(location)
const id = location.pathname.split("/")[2]; //it [2] cause remember
const [slideNumber, setSlideNumber] = useState(0);
const [open, setOpen] = useState(false);
const [openModal, setOpenModal] = useState(false)
const { data, loading, error } = useFetch(`/hotels/find/${id}`);
const {user} = useContext(AuthContext)
const navigate = useNavigate()
const { dates, options } = useContext(SearchContext);

const MILLISECONDS_PER_DAY = 1000 * 60 * 60 * 24;
function dayDifference(date1, date2) {
  const timeDiff = Math.abs(date2.getTime() - date1.getTime());
  const diffDays = Math.ceil(timeDiff / MILLISECONDS_PER_DAY);
  return diffDays;
}
```

Figure 7.1.4. Hooks and Day Difference calculations

This code handles an image slide and displays hotel images. The handleMove function changes the current image based on the direction ("l" for left or any other for right). It wraps around when reaching the first or last image. The JSX part maps through data.photos and shows each photo inside a wrapper. When a photo is clicked, it triggers the handleOpen function to likely open the image in a larger view or modal.

The logic behind this code is to create a simple image slider for hotel photos. The handleMove function updates the currently displayed image index (slideNumber). If the direction is left ("l") and the current slide is the first one (index 0), it jumps to the last image (index 5). Otherwise, it moves one step back. Similarly, if moving right from the last image (index 5), it wraps around to the first; otherwise, it moves forward by one. The image display part uses map() to loop through the photos array and shows each image. Clicking an image triggers handleOpen(i), which likely sets the current image for a full view or slider.

```
const handleOpen = (i) => {
  setSlideNumber(i);
  setOpen(true);
};
const handleMove = (direction) => {
  let newSlideNumber;
  if (direction === "l") {
    newSlideNumber = slideNumber === 0 ? 5 : slideNumber - 1;
  } else {
    newSlideNumber = slideNumber === 5 ? 0 : slideNumber + 1;
  }
  setSlideNumber(newSlideNumber);
};
```

Figure 7.1.5. Sliding Image Logic 1

```
<div className="hotelImages">
  {data.photos?.map((photo, i) => (
    <div className="hotelImgWrapper">
      <img
        onClick={() => handleOpen(i)}
        src={photo}
        alt=""
        className="hotelImg"
      />
    </div>
  ))}
</div>
```

Figure 7.1.6. Sliding Image Logic 2

This code shows how state management works in React using Context and Reducers. It sets up a context provider, which is like a central place to store and share data across multiple components without passing props manually at every level [25]. By using useReducer, it handles more complex state than useState. This method is a helpful way of updating same shared data across the website at the same time. Since I am using a setup where user will be giving some input about the city, destination and dates, this method is super-efficient to keep track at search inputs and updates user data at every stage of the website at the same time.

```
export const SearchContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(SearchReducer, INITIAL_STATE);

  return (
    <SearchContext.Provider
      value={{
        city: state.city,
        dates: state.dates,
        options: state.options,
        dispatch,
      }}
    >
      {children}
    </SearchContext.Provider>
  );
};
```

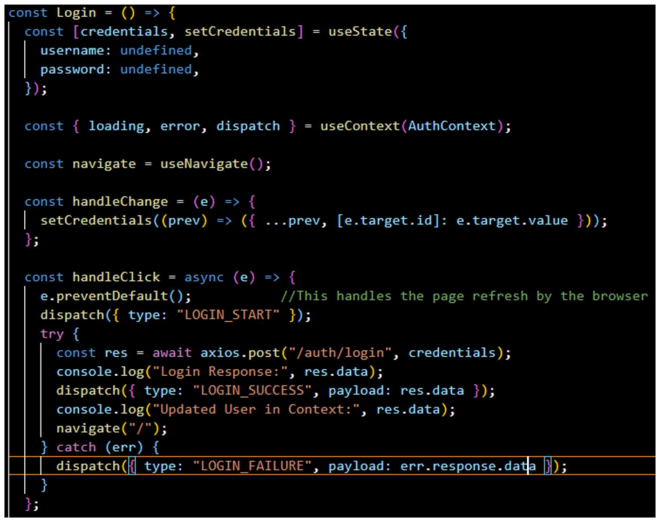
Figure 7.1.7. React Context

```
const SearchReducer = (state, action) => {
  switch (action.type) {
    case "NEW_SEARCH":
      return action.payload;
    case "RESET_SEARCH":
      return INITIAL_STATE;
    default:
      return state;
  }
};
```

Figure 7.1.8. React Reducer

This code handles user login in a React app using state, context, and navigation. It manages input fields for username and password using useState, updates them on change, and sends the login request when the user submits the form. The AuthContext is used to manage authentication

state, triggering actions like starting the login process, handling success by storing user data, or dealing with errors. If login succeeds, it redirects the user to the homepage. This approach helps keep login logic organized and allows different parts of the app to respond to authentication changes.



```
const Login = () => {
  const [credentials, setCredentials] = useState({
    username: undefined,
    password: undefined,
  });

  const { loading, error, dispatch } = useContext(AuthContext);

  const navigate = useNavigate();

  const handleChange = (e) => {
    setCredentials((prev) => ({ ...prev, [e.target.id]: e.target.value }));
  };

  const handleClick = async (e) => {
    e.preventDefault();           //This handles the page refresh by the browser
    dispatch({ type: "LOGIN_START" });
    try {
      const res = await axios.post("/auth/login", credentials);
      console.log("Login Response:", res.data);
      dispatch({ type: "LOGIN_SUCCESS", payload: res.data });
      console.log("Updated User in Context:", res.data);
      navigate("/");
    } catch (err) {
      dispatch({ type: "LOGIN_FAILURE", payload: err.response.data });
    }
  };
```

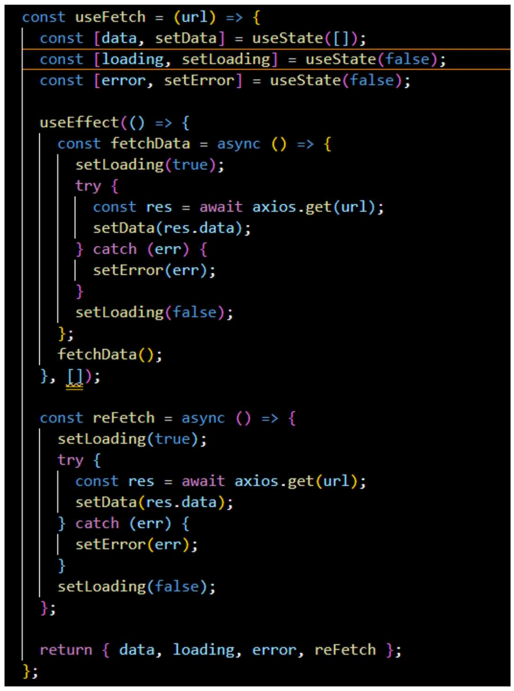
Figure 7.1.9. Login Logic

The AuthReducer function manages the authentication state in a Redux-style reducer. It handles four key actions: starting a login process (LOGIN_START), successfully logging in a user (LOGIN_SUCCESS), handling login failure (LOGIN_FAILURE), and logging out the user (LOGOUT). Each action updates the state accordingly—either setting a loading flag, storing the user data upon success, or recording an error message on failure. By default, the reducer returns the current state if no recognized action is dispatched.

```
const AuthReducer = (state, action) => {
  switch (action.type) {
    case "LOGIN_START":
      return {
        user: null,
        loading: true,
        error: null,
      };
    case "LOGIN_SUCCESS":
      return {
        user: action.payload,
        loading: false,
        error: null,
      };
    case "LOGIN_FAILURE":
      return {
        user: null,
        loading: false,
        error: action.payload,
      };
    case "LOGOUT":
      return {
        user: null,
        loading: false,
        error: null,
      };
    default:
      return state;
  }
};
```

Figure 7.1.10. Authentication State

The useFetch hook is designed to fetch data from a given URL using Axios. It manages three state variables: data for storing the fetched content, loading to indicate the fetching status, and error to capture any issues that occur during the process. The useEffect hook is used to initiate the data fetch when the component first renders. It triggers an async function to make the request and updates the state accordingly. The hook also includes a reFetch function, allowing the data to be reloaded manually. The hook returns the current data, loading status, error, and the reFetch function for further use.



```
const useFetch = (url) => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(false);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      try {
        const res = await axios.get(url);
        setData(res.data);
      } catch (err) {
        setError(err);
      }
      setLoading(false);
    };
    fetchData();
  }, []);

  const reFetch = async () => {
    setLoading(true);
    try {
      const res = await axios.get(url);
      setData(res.data);
    } catch (err) {
      setError(err);
    }
    setLoading(false);
  };

  return { data, loading, error, reFetch };
};
```

Figure 7.1.11. UseEffect Hook

The useEffect hook in this code handles currency conversion based on selected fromCurrency and toCurrency values, using exchange rates stored in the quotes object. And the updated exchange rates are fetched using API keys from a third-party website. It checks three conditions: if both currencies are USD, if neither currency is USD, and if only one currency is USD. Depending on these conditions, it calculates the conversion rate by either directly using USD as a reference or by computing the relative exchange rate between two non-USD currencies. The result is then multiplied by the input amount, rounded to two decimal places, and stored in the convertedAmount state. This effect re-runs whenever the amount, fromCurrency, toCurrency, or quotes change.

```
useEffect(() => {
  if (fromCurrency === "USD" && quotes[`USD${toCurrency}`]) {
    const rate = quotes[`USD${toCurrency}`];
    setConvertedAmount((amount * rate).toFixed(2));
  } else if (fromCurrency !== "USD" && toCurrency !== "USD") {
    const fromRate = quotes[`USD${fromCurrency}`];
    const toRate = quotes[`USD${toCurrency}`];
    if (fromRate && toRate) {
      const rate = toRate / fromRate;
      setConvertedAmount((amount * rate).toFixed(2));
    }
  } else if (fromCurrency !== "USD" && toCurrency === "USD") {
    const fromRate = quotes[`USD${fromCurrency}`];
    if (fromRate) {
      setConvertedAmount((amount / fromRate).toFixed(2));
    }
  }
}, [amount, fromCurrency, toCurrency, quotes]);
```

Figure 7.1.12. Currency Logics

```
useEffect(() => {
  const fetchRates = async () => {
    try {
      const res = await axios.get(API_URL, {
        params: {
          access_key: API_KEY,
          currencies: "AUD,EUR,GBP,PLN,INR,CAD,JPY,CNY,CHF,HKD,SGD",
        },
      });

      if (res.data && res.data.quotes) {
        setQuotes(res.data.quotes);
      }
    } catch (error) {
      console.error("Error fetching exchange rates:", error);
    }
  };

  fetchRates();
}, []);
```

Figure 7.1.12. Fetching Exchange Rates

This logic is fetching Airport data like the flight statuses using a third-party website and by sending their hosted API KEY when used select the Airport Status option in the Flight page.

```
if (apiChoice === "airportsLive") {
  const selectedIATA = airportData[selectedAirport]?.iata;
  endpoint = `https://aerodatabox.p.rapidapi.com/flights/airports/iata/${selectedIATA}
  ?offsetMinutes=-120&durationMinutes=720&withLeg=true&direction=Both&withCancelled
  =true&withCodeshared=true&withCargo=true&withPrivate=true&withLocation=false`;

  const res = await axios.get(endpoint, {
    headers: {
      "x-rapidapi-host": "aerodatabox.p.rapidapi.com",
      "x-rapidapi-key": rapidApiKey,
    },
  });

  setResults([
    ...(res.data?.departures || []),
    ...(res.data?.arrivals || []),
  ]);
```

Figure 7.1.13. API choices

This following snippet code renders a list of departed flights, displaying flight details for each entry. It starts by filtering the results array to only include flights whose status is “departed” (case-insensitive). For each matching flight, it determines the relevant flight information, either from the main item or from the first leg of the flight if available. Rendered flight details included the airline name, flight number, departure and arrival airports, scheduled times, and the flight's status. Each flight's details are displayed with a list titled “departed”. Logics for the list of “Expected flights” are the exact opposite of “departed” list.

Figure 7.1.14. Fetching Flight Status

This following code is for creating a button called Hamburger. This button has 3 more buttons which pops up when user clicks on this hamburger button. These 3 buttons redirect the page to the map page, contact page and the weather page. Same logic is used here for navigation. And "useNavigate" is implemented from "react-router-dom" libraray.

Figure 7.1.15. Hamburger Button Navigation

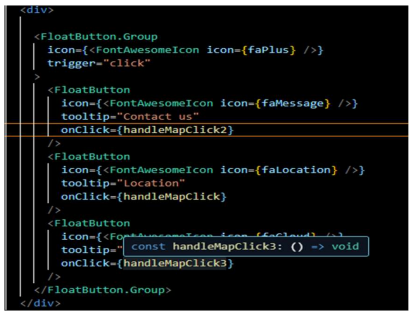


Figure 7.1.16. Hamburger Menu Setup

The FeaturedProperties component fetches a list of featured hotels using the useFetch hook (already described in Figure 7.1.11), displaying up to four properties at a time. While the data is loading, it shows a loading message. Once the data is retrieved, each hotel is displayed in a card with details such as an image, name, city, and starting price. If available, the hotel's rating is also shown with an "Excellent" label. Clicking on a hotel redirects the user to the hotel's detailed page using useNavigate. The component dynamically loads and renders this information in a user-friendly format.

```
export const FeaturedProperties = () => {
  const navigate = useNavigate();
  const { data, loading } = useFetch("/hotels?featured=true&limit=4");

  return (
    <div className="fp">
      {loading ? (
        "Loading"
      ) : (
        data.map((item) => (
          <div
            className="fpItem"
            key={item._id}
            onClick={() => navigate(`/hotelDetails/${item._id}`)}
            style={{ cursor: "pointer" }}
          >
            <img src={item.photos[0]} alt="" className="fpImg" />
            <span className="fpName">{item.name}</span>
            <span className="fpCity">{item.city}</span>
            <span className="fpPrice">Starting from €{item.cheapestPrice}</span>
            {item.rating && (
              <div className="fpRating">
                <button>{item.rating}</button>
                <span>Excellent</span>
              </div>
            )}
          </div>
```

Figure 7.1.17. Fetching Hotel Details

This useEffect hook retrieves the user's current geographical location using the browser's navigator.geolocation API. If geolocation is available, it fetches the latitude and longitude, stores them in the currentLocation state, and then calls a function (fetchWeather) to fetch the weather data for that location. If there's an error retrieving the location, an error message is logged to the console. If geolocation is not supported by the browser, another error message is logged. The effect runs only once when the component mounts, as indicated by the empty dependency array ([]).

```
useEffect(() => {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
      (position) => {
        const location = {
          lat: position.coords.latitude,
          lng: position.coords.longitude,
        };
        setCurrentLocation(location);
        fetchWeather(location);
      },
      (error) => {
        console.error('Error fetching location:', error);
      }
    );
  } else {
    console.error('Geolocation is not supported by this browser.');
  }
}, []);
```

Figure 7.1.18. Google Map Logic

The fetchWeather function asynchronously fetches weather data based on a given location. It constructs a weatherUrl (which is currently a placeholder), sends a request to this URL using the fetch API, and waits for the response. Once the data is received, it checks if the response was successful. If the fetch is successful, the weather data is stored in the weatherData state. If the response contains an error, it logs the error message. In case of any issues during the fetch operation, such as network problems, an error message is logged to the console.

```
const fetchWeather = async (location) => {
  const weatherUrl = `************************`;
  try {
    const response = await fetch(weatherUrl);
    const data = await response.json();
    if (response.ok) {
      setWeatherData(data);
    } else {
      console.error('Weather fetch failed:', data.message);
    }
  } catch (error) {
    console.error('Error fetching weather:', error);
  }
};
```

Figure 7.1.19. Weather Data 1

This “useEffect()” checks if the map is loaded before fetching the current location otherwise it throws an error. Then it checks if a destination is received or not. If all the conditions are satisfied, then it moves to the “handleRouteToCoords” function with a prop “coords” which contains the co-ordinates (Longitude and Latitude).

```
useEffect(() => {
  if (isLoaded && currentLocation && destinationCoords) {
    handleRouteToCoords(destinationCoords);
  }
}, [currentLocation, destinationCoords]);

const handleRouteToCoords = (coords) => {
  const directionsService = new window.google.maps.DirectionsService();
  directionsService.route(
    {
      origin: currentLocation,
      destination: coords,
      travelMode: window.google.maps.TravelMode.DRIVING,
    },
    (result, status) => {
      if (status === 'OK') {
        setDirectionsResponse(result);
      } else {
        console.error(`Route failed: ${status}`);
      }
    }
  );
};
```

Figure 7.1.20. Navigation

The handleSearch function is an asynchronous function that handles searching for tourist places near a specified city. It starts by setting the loading state to true, clearing any existing error messages, and resetting the places array. It then sends a request to the OpenTripMap API to fetch the geographical coordinates (latitude and longitude) of the city. Using these coordinates, it sends another request to fetch nearby “interesting places” within a 5-kilometer radius. If both requests are successful, the list of nearby places is saved in the places state. If any error occurs during the process, it logs the error and sets an error message prompting the user to try another location.

```
const handleSearch = async () => {
  setLoading(true);
  setError("");
  setPlaces([]);

  try {
    const geoRes = await axios.get(
      `https://api.opentripmap.com/0.1/en/places/geoname`,
      {
        params: {
          name: city,
          apikey: API_KEY,
        },
      }
    );

    const { lat, lon } = geoRes.data;

    const nearbyRes = await axios.get(
      `https://api.opentripmap.com/0.1/en/places/radius`,
      {
        params: {
          radius: 5000,
          lon,
          lat,
          kinds: "interesting_places",
          format: "json",
          limit: 70,
          apikey: API_KEY,
        },
      }
    );

    setPlaces(nearbyRes.data);
  } catch (err) {
    console.error(err);
    setError("Could not fetch tourist places. Try another location.");
  }
```

Figure 7.1.21. Tourist Places

Finally, my Index.js initializes the React application by rendering the App component into the root element of the HTML document. It wraps the App component with two context providers: AuthContextProvider and SearchContextProvider. These providers manage authentication and search-related states globally across the app. The use of React.StrictMode helps identify potential problems in the app during development [36]. The root.render method is used to mount the entire React component tree to the DOM.

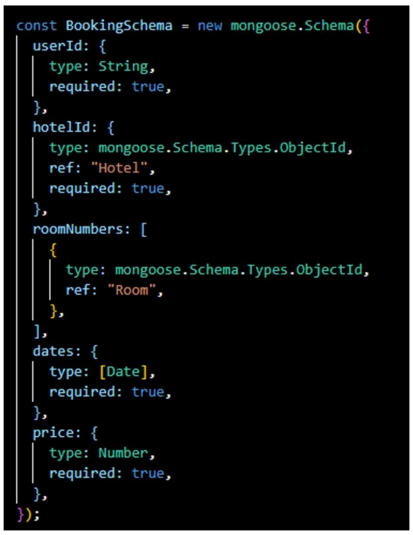
```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <AuthContextProvider>
      <SearchContextProvider>
        <App />
      </SearchContextProvider>
    </AuthContextProvider>


  </React.StrictMode>
);
```

Figure 7.1.22. Index.js

7.2 Backend

My backend is built on purely Node.js and supports all the logics build in the front-end. I have database connections, database schema for different tables with specific name like hotels, bookings, users. Also, it has the routes for different URL to perform CRUD operations requested by the front-end and manage data in the database.

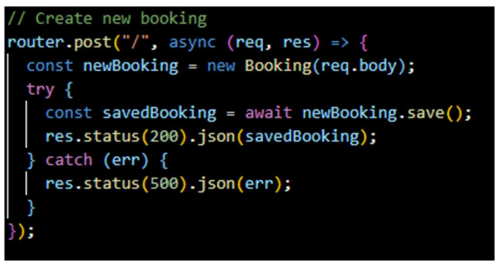
I have 5 schemas: Booking, Users, Hotels, Rooms, Reviews. The structure of all the schemas is pretty much same. This following snippet code is for Booking schema. It includes several fields: userId (a string to store the user's ID), hotelId (an ObjectId that references the Hotel model, indicating the hotel being booked), roomNumbers (an array of ObjectIds referencing the Room model, indicating the rooms being booked), dates (an array of dates representing the booking period), and price (a number representing the total cost of the booking). All the schemas are inter-connected and share information. The schema enforces that certain fields, such as userId, hotelId, dates, and price, are required. The schema is then used to create a Mongoose model named Booking.



```
const BookingSchema = new mongoose.Schema({
  userId: {
    type: String,
    required: true,
  },
  hotelId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Hotel",
    required: true,
  },
  roomNumbers: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Room",
    },
  ],
  dates: {
    type: [Date],
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
});
```

Figure 7.2.0. Schema Example

This code defines a POST route for creating a new booking in the system. When a request is made for this route, it extracts the booking data from the request body (req.body) and creates a new Booking instance using that data. The booking is then saved to the database asynchronously. If the save operation is successful, the saved booking is returned in response with a 200-status code. If an error occurs during the process (e.g., if the database operation fails), a 500-status code is returned along with the error message.



```
// Create new booking
router.post("/", async (req, res) => {
  const newBooking = new Booking(req.body);
  try {
    const savedBooking = await newBooking.save();
    res.status(200).json(savedBooking);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

Figure 7.2.1. Post Request

This GET route retrieves all bookings made by a specific user, identified by the id parameter in the URL. It first queries the Booking collection for records matching the given userId, also populating the related hotelId for additional hotel details. Then, for each booking, it resolves the specific room numbers by looking up the corresponding sub-rooms within the Room collection. These resolved room numbers are collected and added to the final response. The result is an enriched list of bookings that includes both the original booking data and the readable room numbers, which is then sent back to the client with a 200 status. If any part of the process fails, a 500 status is returned with the error information.

```
router.get("/user/:id", async (req, res) => {
  try {
    const bookings = await Booking.find({ userId: req.params.id }).populate("hotelId");

    const enrichedBookings = await Promise.all(
      bookings.map(async (booking) => {
        const resolvedRoomNumbers = [];

        for (const subRoomId of booking.roomNumbers) {
          const room = await Room.findOne({ "roomNumbers._id": subRoomId });

          if (room) {
            const foundSubRoom = room.roomNumbers.find(
              (r) => r._id.toString() === subRoomId.toString()
            );
            if (foundSubRoom) {
              resolvedRoomNumbers.push(foundSubRoom.number);
            }
          }
        }

        return {
          ...booking.toObject(),
          resolvedRoomNumbers,
        };
      })
    );

    res.status(200).json(enrichedBookings);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

Figure 7.2.2. Get Request

The getAllHotels function handles fetching a filtered list of hotels based on query parameters from the request. It processes optional parameters for minimum and maximum price, as well as a limit on the number of results. Any additional query parameters are treated as filters (e.g. city,

type), while min, max, and limit are removed from the filter object. The price range is then added to the filter using MongoDB's $gte (Greater than) and $lte (Less than) operators. Finally, it queries the Hotel collection using these filters and applies the result limit if specified. The resulting list of hotels is returned with a 200 status, and any errors are passed to the error-handling middleware via next().

```
export const getAllHotels = async (req, res, next) => {
    try {
        // Convert query parameters to numbers
        const minPrice = parseInt(req.query.min) || 1;
        const maxPrice = parseInt(req.query.max) || 999;
        const limit = parseInt(req.query.limit) || 0; // Default 0 means no limit

        // Build filter object dynamically
        const filters = { ...req.query };
        delete filters.min;
        delete filters.max;
        delete filters.limit;

        // Add price range condition
        filters.cheapestPrice = { $gte: minPrice, $lte: maxPrice };

        // Fetch hotels with filter and limit
        const hotels = await Hotel.find(filters).limit(limit);

        res.status(200).json(hotels);
    } catch (err) {
        next(err);
    }
};
```

Figure 7.2.3. Price Filtering Logic

The countByType function returns the total number of hotels for each specific type—Hotel, Resort, Apartment, and Airbnb. It uses countDocuments to query the Hotel collection for each type individually, gathering the total count for each one. These counts are then structured into an array of objects, with each object containing a type and its corresponding count. The result is sent back in the response with a 200-status code. If any error occurs during the process, it is passed to the error handler using next().

```
export const countByType = async(req, res, next)=>{
    try{
        const hotelCount =await Hotel.countDocuments({type:"Hotel"});
        const resortCount =await Hotel.countDocuments({type:"Resort"});
        const apartmentCount =await Hotel.countDocuments({type:"Apartment"});
        const airbnbCount = await Hotel.countDocuments({type:"Airbnb"});

        res.status(200).json([
            {type:"Hotel", count: hotelCount},
            {type:"Resort", count: resortCount},
            {type:"Apartment", count: apartmentCount},
            {type:"Airbnb", count: airbnbCount},
        ]);
    }catch(err){
        next(err);
    }
```

Figure 7.2.4. Fetching Hotel Types

The createError function is a simple utility used to generate custom error objects in a consistent format. It takes a status code and a message as arguments, creates a new Error instance, attaches the status and message to it, and then returns the customized error. This helps standardize error handling across the application by making it easy to throw or pass detailed error responses.

```
export const createError = (status, message)=>{
    const err = new Error()
        err.status = status;
        err.message = message;
        return err;
}
```

Figure 7.2.5. Error Handler

The register function handles user registration by securely creating a new user account. It first hashes the provided password using bcrypt, which involves generating a unique salt and applying a cryptographic hash to the password—making it resistant to brute-force and rainbow table attacks. A new User instance is then created with the username, email, hashed password, and an

optional admin flag. The user is saved to the database, and a success message is returned with a 200 status. Any errors encountered are passed to the error-handling middleware via next().

```
export const register = async(req, res, next)=>{
    try{
        const salt = bcrypt.genSaltSync(10);
        const hash = bcrypt.hashSync(req.body.password, salt);
        const newUser = new User({
            username:req.body.username,
            email:req.body.email,
            password:hash,
            isAdmin:  req.body.isAdmin ?? false,
        })

        await newUser.save()
        res.status(200).send("User has been created. ")
    }catch(err){
        next(err)
    }
}
```

Figure 7.2.6. Register User

The login function manages user authentication by verifying credentials and issuing a secure token. It begins by searching for a user by username; if not found, it triggers a 404 error. If the user exists, it uses bcrypt.compare to securely validate the provided password against the stored hashed password. On a successful match, a JWT token is generated containing the user's ID and admin status. This token is then stored in an httpOnly cookie—which is a security feature that prevents client-side scripts from accessing the cookie, helping protect against attacks. Finally, the server responds with user details (excluding the password), confirming successful login. Any errors are forwarded using the next() middleware.

```
export const login = async(req, res, next)=>{
    try{
        const user = await User.findOne({username:req.body.username})
        if(!user) return next(createError(404, "User not found!"))

        const isPasswordCorrect = await bcrypt.compare(req.body.password, user.password)

        if(!isPasswordCorrect) return next(createError(400, "Wrong password or Username!"));

        // if password is correct, we gonna create a new token here:
        const token = jwt.sign({id:user._id, isAdmin: user.isAdmin }, process.env.JWT);

        // this will hide the password when login is successful and shows the username and oth
        const {password, isAdmin, ...otherDetails} = user._doc;
        res.cookie("access_token", token, {
            httpOnly: true,
        }).status(200).json({...otherDetails});
    }catch(err){
        next(err)
    }
}
```

Figure 7.2.7. Login User

Formatted: Justified

The verifyToken function acts as middleware to protect routes by checking if the user is authenticated. It retrieves the JWT token from the access_token cookie, and if it's missing, it returns a 401-error indicating the user is not authenticated. If the token exists, it uses jwt.verify to validate it against the secret key. If the token is invalid or expired, a 403 error is triggered. On successful verification, the decoded user information is attached to the 'req' object for use in subsequent middleware or route handlers. This ensures that only users with a valid token can access protected resources.

```
export const verifyToken = (req, res, next) =>{
    const token = req.cookies.access_token;
    if(!token){
        return next(createError(401, "You are not authenticated!"))
    }
    jwt.verify(token, process.env.JWT, (err, user)=>{
        if(err) return next(createError(403, "Token is not Valid!"));
        req.user = user;
        next();

    });
};
```

Figure 7.2.8. Token Verification

The verifyUser function is a middleware used to authorize access based on user identity. It first calls verifyToken to ensure the request includes a valid JWT. Once verified, it checks whether the user is an admin or if the user's ID matches the ID in the request parameters. If either condition is true, the request proceeds; otherwise, a 403 error is returned, indicating the user is not authorized to access the resource.

```
export const verifyUser = (req, res, next)=>{
    verifyToken(req, res, (err)=>{
        console.log("req.user.id:", req.user.id);
        console.log("req.params.userId:", req.params.userId);
        if (err) return next(err);

        if(req.user.isAdmin || req.user.id === req.params.id){
            next()
        }else{
            return next(createError(403, "You are not authorized!"));

        }
    })
}
```

Figure 7.2.9. Middleware Example

This is in my index.js where it checks the connection with the MongoDB. The MongoDB URL is stored in .env file. If the connection is successful, it logs a message in the terminal otherwise it shows connection failure message.

```
const connect = async ()=>{
    try {
        await mongoose.connect(process.env.MONGO);
        console.log("Connected to mongoDB!")
    } catch (error) {
        throw error
    }
};

mongoose.connection.on("disconnected", ()=>{
    console.log("mongoDB Disconnected!")
})
```

Figure 7.2.10. MongoDB connection

This part sets up the core of the Express app. It enables CORS so the frontend can talk to the backend, handles cookies and JSON data, and wires up routes for things like auth, users, hotels, rooms, bookings, and reviews. There's also a centralized error handler that catches anything that goes wrong and sends back a helpful error response. Finally, the server starts listening on port 8081, and once it's up, it connects to the database and logs a message to confirm everything's running smoothly.

```
app.use(cors())
app.use(cookieParser())
app.use(express.json())
app.use("/api/auth", authRoute);
app.use("/api/users", usersRoute);
app.use("/api/hotels", hotelsRoute);
app.use("/api/rooms", roomsRoute);
app.use("/api/bookings", bookingsRoute);
app.use("/api/reviews", reviewsRoute);

app.use((err, req, res, next)=>{
    const errorStatus = err.status || 500
    const errorMessage = err.message || 'Something went wrong'

    return res.status(errorStatus).json({
        success: false,
        status: errorStatus,
        message: errorMessage,
        stack: err.stack,
    });
});

app.listen(8081, ()=>{
    connect()
    console.log("Connected to backend!")
});
```

Figure 7.2.11. index.js

The updateHotelRating function calculates and updates a hotel's rating using the Bayesian Average method, which helps provide a more balanced rating, especially for hotels with fewer reviews. It first retrieves all reviews for the specified hotel and calculates the average rating (R) based on those reviews. If the hotel has no reviews, it sets the rating to 0. Then, it calculates the

global average rating (C) by considering all reviews across hotels. The function uses a minimum review threshold (m) to weigh the hotel's own average rating against the global average, resulting in a more accurate, fairer rating. Finally, the computed Bayesian rating is rounded to two decimal places and updated in the hotel's database record.

```
// Bayesian Average method
const updateHotelRating = async (hotelId) => {
  const hotelReviews = await Reviews.find({ hotelId });
  const v = hotelReviews.length;

  if (v === 0) {
    await Hotel.findByIdAndUpdate(hotelId, { rating: 0 });
    return;
  }

  const R = hotelReviews.reduce((sum, review) => sum + review.rating, 0) / v;
  console.log(`Hotel ${hotelId} average rating (R): ${R}`);

  const allReviews = await Reviews.find();
  const C =
    allReviews.reduce((sum, review) => sum + review.rating, 0) /
    allReviews.length;

  console.log(`Global average rating (C): ${C}`);
  const m = 5; // minimum number of reviews
  console.log(`Minimum reviews required (m): ${m}`);

  const bayesianRating = (v / (v + m)) * R + (m / (v + m)) * C;
  console.log(`Bayesian rating for hotel ${hotelId}: ${bayesianRating}`);

  await Hotel.findByIdAndUpdate(hotelId, {
    rating: parseFloat(bayesianRating.toFixed(2)),
  });
};
```

Figure 7.2.12. Bayesian Reviews

8 Ethics

The GlobeTrek project aims to provide valuable service to travellers while ensuring ethical standards are upheld throughout the development and operation of the platform. One of the main ethical considerations is user privacy. Since the platform collects user data such as usernames, emails, booking details, and card numbers, it is important to ensure that personal information is securely stored and never misused. By keeping that in mind, I have implemented standard security measures such as encryption and secure password storage. Furthermore, since the platform integrates third-party APIs for services like weather updates, currency exchange rates, and flight status, ethical use of these APIs is crucial. This includes ensuring the data provided through these external services is accurate. For this purpose, I utilized some of the well-known and transparent third-party APIs like Google Maps and OpenWeather API.

To make everything transparent to the customers, this platform also allows users to rate their hotels and leave reviews, which are managed using the Bayesian Review system. This method ensures fair review for both parties, hotel owners and the future customers.

Finally, while the platform offers location-based services like maps and nearby attractions, user consent is obtained while fetching their coordinates from the browser. This ethical responsibility of ensuring that users are fully informed while sharing their data.

Being an engineer, I am always committed to protecting the security of the personal data of others, specifically when developing such a website, which stores the personal data of the customers. And my website is built by considering those security measures, and any future developments will always prioritize customers information security before anything.

9 Conclusion

In conclusion, I'm proud to say that I've achieved most of the goals I set out for this project. I have a fully functional website that can be used to make hotel bookings, and it features several helpful APIs to enhance the overall travel experience for the users.

This journey was incredibly rewarding and educational. When I started, I had just a little knowledge about the MERN stack and barely any experience in building such a full-scale application. Going through the entire process- from planning and designing to building, I have confidence for future development of this project. One idea is to introduce machine learning algorithms for personalized recommendations and price predictions of the hotels by analysing a user's previous booking history. This could really help users make quicker and better decisions when choosing a place to stay.

Overall, this project has been a great learning experience and a big step forward in my development journey.

Some Glimpse of my Final Product:

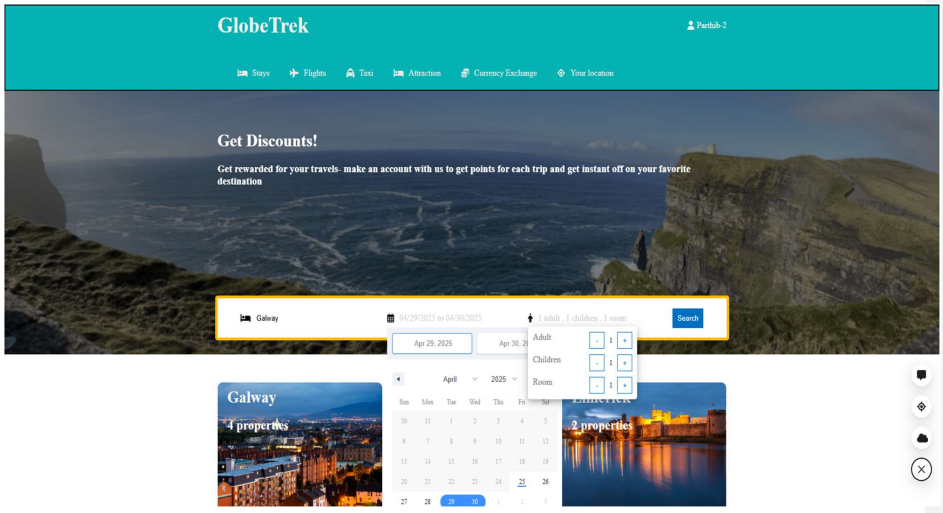


Figure 9.1. Home Page

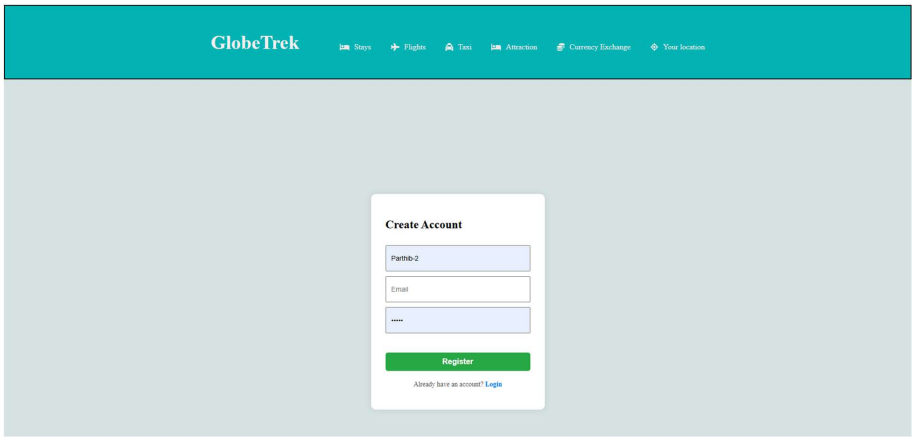


Figure 9.2. Register Page

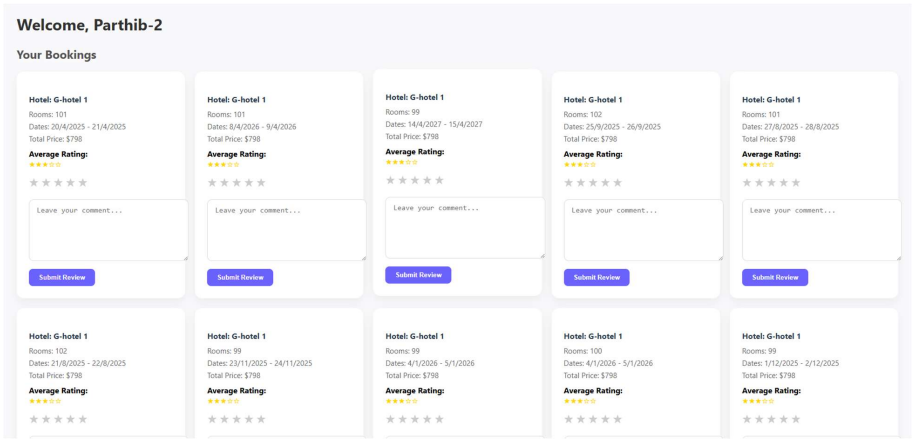


Figure 9.3. User Dashboard

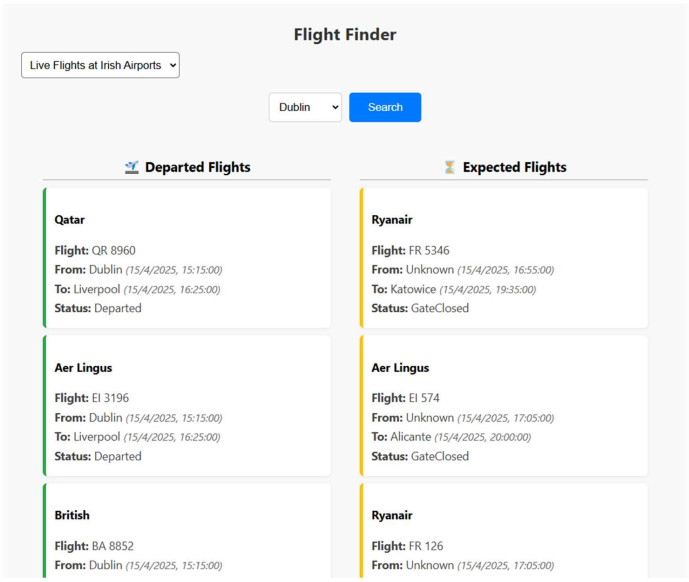


Figure 9.4. Flight Status

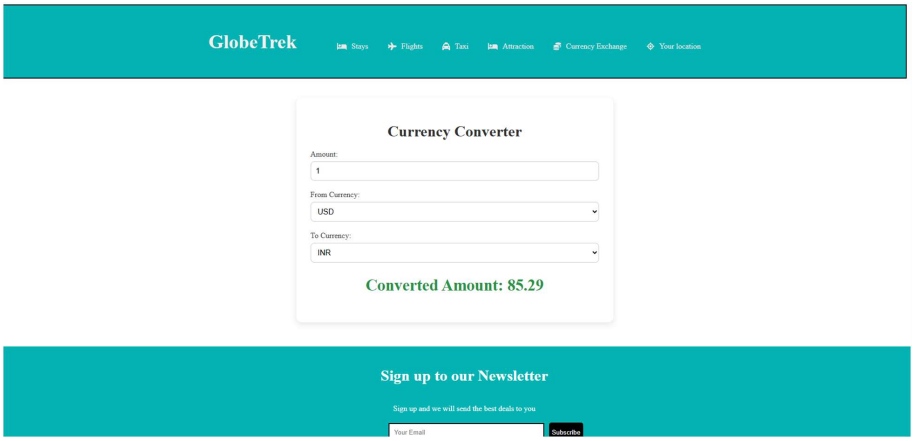


Figure 9.5. Currency Exchange Rates

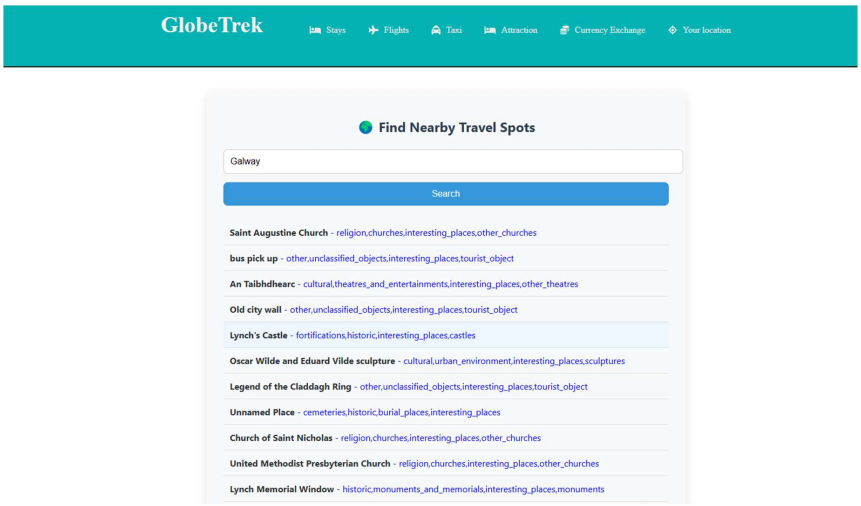


Figure 9.6. Nearby Tourist Places

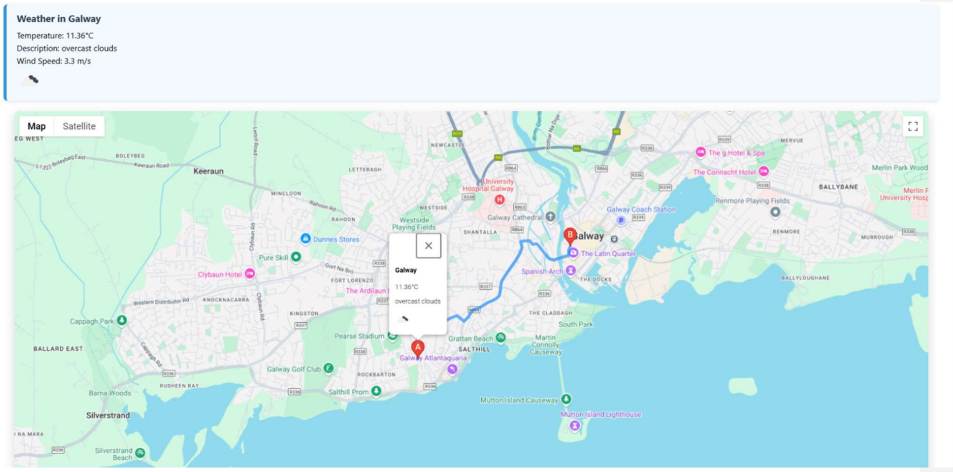


Figure 9.7. Maps Navigation and Weather

11 References

[1] GeeksForGeeks, "Express routing in MERN stack", [Online], Available: https://www.geeksforgeeks.org/express-routing-in-mern-stack/

[2] GeeksForGeeks, "What is Express", [Online], Available: What is Express? | GeeksforGeeks

[3] freeCodeCamp(), “Introduction to Mongoose for MongoDB”, [Online], Available: https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/

[4] GeeksForGeeks, "What is Elastic Compute Cloud (EC2)", [Online], Available: What is Elastic Compute Cloud (EC2)? | GeeksforGeeks

[5] Medium, "Routing and Navigation in React.js: A comprehensive guide with Examples", [Online], Available: Routing and Navigation in React.js: A Comprehensive Guide with Examples | by Pawan Kumar | Medium

[6] OpenWeatherMap, "Science", [Online], Available: OpenWeatherMap API — Public APIs

[7] GeeksForGeeks, "JSON Web Token (JWT)", [Online], Available: JSON Web Token (JWT) | GeeksforGeeks

[8] freeCodeCamp(), “How to Hash Passwords with bcrypt in Node.js”, [Online], Available: How to Hash Passwords with bcrypt in Node.js

[9] React, “React”, [Online], Available: React

[10] w3schools, “React Tutorial”, [Online], Available: React Tutorial

[11] Create React App, “Create-React-App”, [Online], Available: Getting Started | Create React App

[12] w3schools, “Node.js Introduction”, [Online], Available: Node.js Introduction

[13] freeCodeCamp(), “What Exactly is Node.js? Explained for Beginners”, [Online], Available: What Exactly is Node.js? Explained for Beginners

[14] GeeksForGeeks, "NodeJS Introduction", [Online], Available: NodeJS Introduction | GeeksforGeeks

[15] freeCodeCamp(), "What Exactly is Node.js and why should you use it? Explained for Beginners", [Online], Available: What exactly is Node.js and why should you use it?

[16] algolia DOCUMENTATION, "Using the Bayesian average in custom ranking", [Online], Available: Using the Bayesian average in custom ranking | Algolia

[17] Font Awesome Docs, "Add Icons with React", [Online], Available: https://docs.fontawesome.com/web/use-with/react/add-icons

[18] React, "Built-in React Hooks", [Online], Available: Built-in React Hooks – React

[19] w3schools, "React Hooks", [Online], Available: React Hooks

[20] GeeksForGeeks, "React Hooks", [Online], Available: React Hooks | GeeksforGeeks

[21] React, "Managing State", [Online], Available: Managing State – React

[22] GeeksForGeeks, "State Management in React – Hooks, Context API and Redux", [Online], Available: State Management in React – Hooks, Context API and Redux | GeeksforGeeks

[23] DEV, "Top 10 State Management Libraries for ReactJS", [Online], Available: Top 10 State Management Libraries for ReactJS - DEV Community

[24] freeCodeCamp(), "State Management in React – When and Where to use State", [Online], Available: State Management in React – When and Where to use State

[25] React, "useContext", [Online], Available: useContext – React

[26] freeCodeCamp(), "React Context API Explained with Examples", [Online], Available: React Context API Explained with Examples

[27] React, "createContext", [Online], Available: createContext – React

[28] w3schools, "React Router", [Online], Available: React Router

[29] React Router, "Installation", [Online], Available: Installation | React Router

[30] AXIOS, “Getting Started”, [Online], Available: Getting Started | Axios Docs

[31] Medium, “A Beginner’s Guide to Using Axios in Node.js: Simplifying HTTP Requests”, [Online], Available: A Beginner’s Guide to Using Axios in Node.js: Simplifying HTTP Requests | by Reggie Cheston | Medium

[32] CodeGreek, “Axios in Node.js: For Making HTTP Requests”, [Online], Available: Axios in Node.js: For Making HTTP Requests | CodeForGeek

[33] GeeksForGeeks, “What is Axios”, [Online], Available: What Is Axios? | GeeksforGeeks

[34] GeeksForGeeks, “What is the use of ORM/ODM libraries like Mongoose in Express applications”, [Online], Available: What is the use of ORM/ODM libraries like Mongoose in Express applications ? | GeeksforGeeks

[35] npm, “bcrypt”, [Online], Available: bcrypt - npm

[36] React, “Strict Mode”, [Online], Available: Strict Mode – React