Made a basic hotel reservation website(using reactjs, mongodb, jwt tokens)

1. **Created a backend server**

* Npm init -y(this creates a package.json file)
* Npm I express
* In package.json cwrote type = module(this removes the need of require) using import and export
* Index.js
* Nodemon index.js

1. **Setting up our database**

* **Cloud**.mongodb.com used the free plan
* **Username :padma password:padma**
* U have to change the ip address as the 0 so anyone can access it, but I haven’t done it for this project
* Placed the necessary info inside the .env file
* Then using mongoose for collections, and also to connect to mongodb
* Created a folder called routes

**Auth.js Hotel.js Room.js User.js**

**Created** middlewares

app.use("/api/auth", authRoute);

router.post("/register", register)

router.post("/login", login)

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

//UPDATE

router.put("/:id", verifyUser, updateUser);

//DELETE

router.delete("/:id", verifyUser, deleteUser);

//GET

router.get("/:id", verifyUser, getUser);

//GET ALL

router.get("/", verifyAdmin, getUsers);

app.use("/api/hotels", hotelsRoute);

router.post("/", verifyAdmin, createHotel);

//UPDATE

router.put("/:id", verifyAdmin, updateHotel);

//DELETE

router.delete("/:id", verifyAdmin, deleteHotel);

//GET

router.get("/find/:id", getHotel);

//GET ALL

router.get("/", getHotels);

router.get("/countByCity", countByCity);

router.get("/countByType", countByType);

router.get("/room/:id", getHotelRooms);

app.use("/api/rooms", roomsRoute);

router.post("/:hotelid", verifyAdmin, createRoom);

//UPDATE

router.put("/availability/:id", updateRoomAvailability);

router.put("/:id", verifyAdmin, updateRoom);

//DELETE

router.delete("/:id/:hotelid", verifyAdmin, deleteRoom);

//GET

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

//GET ALL

router.get("/", getRooms);

Model.js we have kept our schema

Certainly! Here's a summary of the project based on the components and functionalities described:

**Project Overview**

The project is a web application for managing and displaying hotels and rooms, including features like hotel listing, room availability, and property type counts. It involves both a backend (server) and a frontend (client) component.

**Backend (Server)**

1. **API Endpoints:**
   * **Hotels:**
     + **GET /hotels**: Fetches a list of hotels, with optional query parameters for filtering by price range and limiting the number of results.
     + **POST /hotels**: Creates a new hotel (requires admin authentication).
     + **PUT /hotels/:id**: Updates an existing hotel (requires admin authentication).
     + **DELETE /hotels/:id**: Deletes a hotel (requires admin authentication).
     + **GET /hotels/countByCity**: Counts hotels by city.
     + **GET /hotels/countByType**: Counts hotels by type (e.g., hotel, apartment, resort).
     + **GET /hotels/:id/room**: Fetches rooms for a specific hotel.
   * **Rooms:**
     + **POST /rooms/:hotelid**: Creates a new room in a specific hotel (requires admin authentication).
     + **PUT /rooms/:id**: Updates room details (requires admin authentication).
     + **PUT /rooms/availability/:id**: Updates room availability.
     + **DELETE /rooms/:id/:hotelid**: Deletes a room from a specific hotel (requires admin authentication).
     + **GET /rooms/:id**: Fetches details of a specific room.
     + **GET /rooms**: Fetches a list of all rooms.
2. **Authentication and Authorization:**
   * **verifyToken**: Middleware to ensure a user is authenticated by checking their JWT.
   * **verifyUser**: Middleware to ensure a user is authorized to access specific resources based on their ID or admin status.
   * **verifyAdmin**: Middleware to ensure only admins can access certain routes.

**Frontend (Client)**

1. **Custom Hook (useFetch):**
   * **Purpose**: Simplifies data fetching from the backend by managing loading states, errors, and data updates.
   * **Functionality**: Uses **axios** to make HTTP GET requests and returns the fetched data, loading state, error state, and a function to re-fetch data.
2. **Components:**
   * **FeaturedProperties**:
     + **Purpose**: Displays featured hotels.
     + **Data Fetching**: Uses **useFetch** to get hotels that are marked as featured and limits the result to 4 hotels.
     + **Rendering**: Maps over the fetched data and displays each hotel with its image, name, city, price, and rating.
   * **PropertyList**:
     + **Purpose**: Displays a list of hotel types with images and counts.
     + **Data Fetching**: Uses **useFetch** to get the count of hotels by type.
     + **Rendering**: Maps over the predefined set of images and displays hotel types with their counts.

**Summary**

* **Backend**: Manages hotel and room data with CRUD operations and provides endpoints for various queries. Includes middleware for authentication and authorization.
* **Frontend**: Uses React to build components that fetch and display data from the backend. Custom hooks simplify data fetching and state management.

This project combines backend API development with frontend React components to create a complete application for managing and displaying hotel information.

**##################################################################################**

**Project Overview**

This project is a web application that utilizes React for the frontend, and it incorporates React Router for client-side routing and context management for state handling. The application includes features like searching for hotels, viewing hotel details, and user authentication.

**Key Components and Features**

1. **React Router Setup**:
   * **BrowserRouter**: Wraps the entire application, enabling client-side routing.
   * **Routes**: Defines the different routes available in the application.
   * **Route**: Maps specific paths to components, handling navigation between different views.
2. **Pages**:
   * **Home**: The landing page of the application where users can perform a search.
   * **List**: Displays search results based on user input from the Home page. It shows a list of hotels filtered by the search parameters (destination, dates, options).
   * **Hotel**: Displays detailed information about a specific hotel. The hotel ID is passed as a URL parameter.
   * **Login**: A page for user authentication.
   * **ThankYou**: A page displayed after certain actions, such as booking or successful login.
3. **Context Management**:
   * **AuthContext**: Manages authentication state and user information across the application. It allows components to access and update authentication-related data.
   * **SearchContext**: Manages the state related to search parameters (destination, dates, options) using a reducer to handle updates and resets.
4. **Functionality**:
   * **Searching**: Users can search for hotels based on destination, dates, and options. The search parameters are managed in **SearchContext** and passed via **React Router**'s state.
   * **Hotel Details**: Users can view detailed information about a specific hotel, accessed through URL parameters.
   * **Authentication**: Provides login functionality and manages user sessions using **AuthContext**.
5. **Components**:
   * **Navbar** and **Header**: Common components used across different pages for consistent navigation and branding.
   * **SearchItem**: Displays individual search results (hotels) on the **List** page.
   * **useFetch**: A custom hook for fetching data from an API.

**Technical Details**

* **React Router**: Manages navigation and routing within the app.
* **React Context API**: Used for global state management related to user authentication and search parameters.
* **React Hooks**: Utilized for managing component state and lifecycle (e.g., **useState**, **useReducer**, **useLocation**).

**Workflow**

1. **User performs a search** on the Home page.
2. **Search parameters** are dispatched to **SearchContext** and passed to the **/hotels** route via **navigate**.
3. **List component** retrieves search parameters using **useLocation** and displays search results.
4. **Hotel details** are viewed on a separate page using route parameters.
5. **User authentication** and session management are handled through **AuthContext**.

**Summary**

This project demonstrates proficiency in building a React-based web application with client-side routing, state management, and context API for handling global state. It includes user authentication, dynamic data fetching, and a user-friendly interface for searching and viewing hotels.

Sure! Let's dive into the **handleSelect** and **handleClick** functions from the **Reserve** component:

**1. handleSelect Function**

javascript

Copy code

const handleSelect = (e) => { const checked = e.target.checked; // True if the checkbox is checked, false if unchecked const value = e.target.value; // ID of the room being selected/deselected // Update selectedRooms state setSelectedRooms( checked ? [...selectedRooms, value] // If checked, add the room ID to selectedRooms : selectedRooms.filter((item) => item !== value) // If unchecked, remove the room ID from selectedRooms ); };

**Purpose**:

* **Selection Management**: Updates the list of selected rooms based on user interactions with checkboxes.

**How it Works**:

1. **Event Handling**:
   * **e.target.checked** gets the current state of the checkbox (checked or unchecked).
   * **e.target.value** gets the ID of the room associated with the checkbox.
2. **State Update**:
   * If the checkbox is checked (**checked** is **true**), the room ID (**value**) is added to the **selectedRooms** array using the spread operator (**[...selectedRooms, value]**).
   * If the checkbox is unchecked (**checked** is **false**), the room ID (**value**) is removed from the **selectedRooms** array using **filter**.

**Outcome**:

* The **selectedRooms** state keeps track of which rooms the user has selected for reservation.

**2. handleClick Function**

javascript

Copy code

const handleClick = async () => { try { await Promise.all( selectedRooms.map((roomId) => { const res = axios.put(`/rooms/availability/${roomId}`, { dates: alldates, // Dates for which the room is being reserved }); return res.data; // Return the response data from each PUT request }) ); setOpen(false); // Close the reservation modal navigate("/thank"); // Redirect to the thank you page } catch (err) { // Handle errors here (e.g., display an error message) } };

**Purpose**:

* **Reservation Processing**: Updates the availability of selected rooms and handles navigation after successful reservation.

**How it Works**:

1. **Updating Room Availability**:
   * **Promise.all**: Executes multiple asynchronous requests concurrently. For each room ID in **selectedRooms**, it sends a **PUT** request to update the room's availability.
   * **Endpoint**: The **PUT** request is made to **/rooms/availability/${roomId}** with a payload containing the dates for which the room is being reserved (**dates: alldates**).
2. **Handling Response**:
   * Waits for all **PUT** requests to complete.
   * Upon success, closes the reservation modal (**setOpen(false)**) and navigates to the **/thank** page using **navigate("/thank")**.
3. **Error Handling**:
   * If any of the **PUT** requests fail, the error is caught in the **catch** block. This is where you can handle errors, such as displaying an error message to the user.

**Outcome**:

* **Success**: Updates the room availability on the server and then redirects the user to a thank you page, signaling the end of the reservation process.
* **Error**: The **catch** block handles any errors that occur during the update process, allowing for appropriate error handling.

In summary:

* **handleSelect**: Manages the state of room selections by adding or removing room IDs based on checkbox interactions.
* **handleClick**: Handles the reservation process by updating room availability on the server and then redirecting the user to a thank you page.