PROJECT : INVOCONNECT

File: `App.jsx`

Purpose

The `App.jsx` component is the main entry point for the InvoConnect frontend, handling routing, authentication, and theming across the app. It wraps the application in context providers for managing authentication and theme preferences and includes error handling with `ErrorBoundary`.

Key Functionalities

1. **Context Providers**:
   - `AuthProvider`: Provides authentication context, making user authentication status and related methods accessible across the app.
   - `ThemeProvider`: Manages the theme (light or dark mode) throughout the app.
2. **Toaster for Notifications**:
   - A `Toaster` instance is created to show notifications within the app, with messages (e.g., success or error notifications) displayed in a consistent style. This component is built with BlueprintJS.
3. **Routes**:
   - **Public Routes**:
     - `/login` (renders `Login` component): For user login.
     - `/register` (renders `Register` component): For user registration.
   - **Protected Routes**:
     - These routes are only accessible by authenticated users with specific roles.
     - `/admin/dashboard` (renders `AdminDashboard`): Restricted to users with an "admin" role.
     - `/business/dashboard` (renders `BusinessOwnerDashboard`): Restricted to users with a "business_owner" role.
   - **Default Redirect**:
     - `/` redirects to `/login` if the user is not authenticated.
   - **404 Not Found Route**:
     - Redirects to a custom `NotFound` component when an invalid route is accessed.
4. **ProtectedRoute Component**:
   - Wraps components to ensure only users with appropriate roles can access them.
   - Passes an `allowedRoles` prop to specify which roles can access a route.

Backend Requirements

The backend will need endpoints and logic for handling authentication, role-based authorization, and user registration. Below are the requirements and expected responses for each:

1. **Authentication**:

- **Endpoint**: `/api/auth/login`
- **Method**: `POST`
- **Data**: `{ username: string, password: string }`
- **Response**:
  - `200 OK` with `{ token: string, role: string }` if credentials are valid.
  - `401 Unauthorized` if credentials are invalid.

Note: The token should be stored on the frontend (e.g., in `AuthContext`) and used to identify the user for role-based access.

2. **Registration**:
   - **Endpoint**: `/api/auth/register`
   - **Method**: `POST`
   - **Data**: `{ username: string, password: string, role: string }`
   - **Response**:
     - `201 Created` with `{ message: "User registered successfully." }`
     - `400 Bad Request` if there’s an issue with the registration data.
3. **Role-Based Access**:
   - The backend should verify the user's role on each protected route:
     - **Admin Role**: Access to `/admin/dashboard`.
     - **Business Owner Role**: Access to `/business/dashboard`.
   - For simplicity, this can be done by including the user's role in the JWT or by checking the role on each route request.
4. **Error Handling**:
   - The backend should return appropriate error messages and status codes. For instance:
     - `401 Unauthorized` if a user without permission tries to access a protected route.
     - `404 Not Found` for invalid endpoints.

The Django developer should configure JWT-based authentication to enable the frontend to authenticate users and manage access based on roles. Additionally, error responses should be structured to ensure the frontend can display relevant messages through the `Toaster`.

File: `main.jsx`

Purpose

The `main.jsx` file is the entry point for the React application. It sets up the root React component, `App`, and applies global error handling and styling configurations.

Key Functionalities

1. **Error Boundary**:
   - Wraps the `App` component with `ErrorBoundary` to catch any errors in the component tree, providing a fallback UI in case of unexpected errors. This ensures the app fails gracefully.
2. **React Root Creation**:
   - `createRoot` initializes React in the DOM by finding the `root` HTML element, creating a root instance, and rendering the app within it.
3. **Strict Mode**:
   - The `React.StrictMode` wrapper is used to highlight potential problems in the application during development, such as deprecated lifecycle methods or unsafe coding patterns. This mode does not affect production but helps catch issues in development.
4. **Global Styles**:
   - Imports `index.css` and Blueprint CSS to apply global styles and design elements consistently throughout the app.

Backend Considerations

This file is strictly for initializing the frontend and doesn’t directly affect backend functionality. However, understanding that this file serves as the starting point of the application can help the Django developer recognize where global components like error boundaries are configured, ensuring error handling is also prioritized in the backend.

File: `ErrorBoundary.jsx`

Purpose

The `ErrorBoundary` component is a React error boundary that catches JavaScript errors in the component tree and displays a user-friendly message instead of causing a full application crash. It also provides the user with an option to reset the error and retry.

Key Functionalities

1. **Error State Management**:
   - **State**: Manages `hasError`, `error`, and `isLoading` states.
     - `hasError`: Indicates if an error has occurred.
     - `error`: Stores the error details.
     - `isLoading`: Allows displaying a loading spinner if needed.
2. **Error Handling**:
   - **getDerivedStateFromError**: This lifecycle method sets `hasError` to `true` and records the error, enabling the component to switch to an error UI.
   - **componentDidCatch**: Catches additional error details and logs them to the console. Ideally, the error details should be sent to an error reporting service or backend API endpoint to keep track of issues.
3. **Reset Error Handler**:
   - **handleResetError**: Resets the `hasError` and `error` states, allowing users to retry or continue using the app without reloading the page. This could be useful after backend fixes or reconnecting if an error is network-related.
4. **Conditional Rendering**:
   - If an error occurs, the component renders a styled `Callout` from BlueprintJS, displaying the error message with a retry button.
   - The `Spinner` component renders if `isLoading` is `true`, indicating that data is being fetched or processed.

Backend Requirements

The backend developer may need to create an endpoint to log errors caught by the frontend, especially for persistent issues that require tracking and analysis. Here are the specifications:

1. **Error Logging Endpoint**:
   - **Endpoint**: `/api/logs/error`
   - **Method**: `POST`
   - **Data**: `{ error: string, errorInfo: object, timestamp: Date, user: string }`
   - **Purpose**: Allows the frontend to send error details, providing insights into issues that users encounter, which can be addressed in future updates.
   - **Response**:
     - `200 OK` if the error was logged successfully.
     - `500 Internal Server Error` if the logging process fails.

By implementing an error logging endpoint, the backend will support consistent monitoring and troubleshooting, which aligns with the frontend's `ErrorBoundary` component.

File: `ThemeToggle.jsx`

Purpose

The `ThemeToggle` component is a button that allows users to switch between light and dark themes in the application. It utilizes the theme context (`ThemeContext`) to access and modify the theme settings.

Key Functionalities

1. **Theme Context**:
   - **useTheme**: A custom hook from `ThemeContext` that provides two properties:
     - `isDark`: A boolean that indicates the current theme mode (true for dark, false for light).
     - `toggleTheme`: A function to switch between dark and light themes.
2. **Button Display**:
   - The button displays an icon and text based on the current theme:
     - **Icon**: Uses BlueprintJS icons — `IconNames.FLASH` for light mode and `IconNames.MOON` for dark mode.
     - **Text**: Displays "Light Mode" when the dark theme is active and "Dark Mode" when the light theme is active.
3. **User Interaction**:
   - **onClick**: The button's click handler calls `toggleTheme`, which updates the theme in the context, affecting the UI across the app.

Backend Considerations

This component primarily impacts the frontend and doesn't require backend support. However, if theme preferences need to persist across sessions, the backend can offer an endpoint to save and retrieve a user's theme setting.

1. **Optional Theme Preference Storage**:
   - **Endpoint**: `/api/user/theme`
   - **Methods**:
     - `POST`: To save the theme preference, `{ user_id: string, theme: "dark" | "light" }`.
     - `GET`: To retrieve the theme preference, returning `{ theme: "dark" | "light" }`.
   - **Purpose**: Ensures the user's theme setting is maintained across logins, providing a personalized experience.
   - **Response**:
     - `200 OK` if the theme preference is successfully saved or retrieved.
     - `401 Unauthorized` if the user is not authenticated.

This optional backend functionality would allow theme preferences to persist for each user.

File: `Login.jsx`

Purpose

The `Login` component renders a login form for user authentication. It collects user credentials and invokes the `login` function from the authentication context (`AuthContext`) to handle login requests.

Key Functionalities

1. **Form Data Management**:
   - **State**:
     - `formData`: Holds `username` and `password` entered by the user.
     - `error`: Stores error messages if the login attempt fails.
   - **handleChange**: Updates `formData` as the user types and clears any previous error messages when a field changes.
2. **Login Submission**:
   - **handleSubmit**: Triggers when the login form is submitted. It:
     - Prevents default form submission behavior.
     - Calls the `login` function from `AuthContext`, passing in `formData`.
     - If login fails, `error` is updated to display an error message in the UI.
3. **Error Handling**:
   - When a login error occurs, the `error` state is populated with an error message. This message displays in a Blueprint `Callout` component with a "danger" intent to visually indicate the issue.
4. **Theme Toggle**:
   - The `ThemeToggle` component allows users to switch between light and dark themes directly from the login page.
5. **BlueprintJS Styling**:
   - Uses Blueprint components for a modern UI:
     - `Card`: Wraps the login form with a card-like appearance.
     - `Button`, `FormGroup`, `InputGroup`: For form elements.
     - `H1`: Renders the "Login" title.
   - Inline styles are used for centering and responsive layout.

Backend Requirements

The backend developer should create an endpoint to handle login requests and return the necessary authentication data. Here’s an outline:

1. **Login Endpoint**:
   - **Endpoint**: `/api/auth/login`
   - **Method**: `POST`
   - **Expected Payload**:

     ```
     {
       "username": "string",
       "password": "string"
     ```

```
}
```

- **Response**:
  - **Success**: Returns a token (JWT or session token) and possibly user details or role information.

    ```
    {
      "token": "jwt_token",
      "user": {
        "id": "integer",
        "username": "string",
        "role": "admin" | "business_owner"
      }
    }
    ```

  - **Error**: Returns an error message (e.g., "Invalid username or password").

2. **Session Management**:
   - If using JWT, the token should be saved in the frontend, either in localStorage or a React context, allowing access to protected routes in the application.
3. **Error Handling**:
   - Ensure meaningful error messages are returned for failed login attempts so they can be displayed in the `Callout` component in `Login.jsx`.

This setup will allow seamless authentication flow between the frontend and backend.

File: `Register.jsx`

Purpose

The `Register` component allows users to create a new account by entering their credentials. It performs validation checks on the input fields and sends the registration data to the authentication context (`AuthContext`) for processing.

Key Functionalities

1. **Form Data Management**:
   - **State**:
     - `formData`: Holds the user's `username`, `password`, and `confirmPassword`.
     - `errors`: An object that stores error messages for form validation.
   - **handleChange**: Updates `formData` based on user input and clears specific errors when the user begins typing.
2. **Validation Logic**:
   - **validateForm**: Checks the validity of user inputs before submission:
     - Ensures that the username is provided and has a minimum length of 3 characters.
     - Validates that the password is provided and has a minimum length of 8 characters.
     - Checks that the password and confirm password fields match.
   - If any validation fails, appropriate error messages are set in the `errors` state.
3. **Registration Submission**:
   - **handleSubmit**: Invoked on form submission. It:
     - Prevents the default form behavior.
     - Calls `validateForm` to check inputs.
     - If valid, generates a unique 4-digit User ID (UID) and invokes the `register` function from `AuthContext`, passing the `username` and `uid`.
     - If registration fails, the error message is stored in the `errors.submit` state.
4. **Error Handling**:
   - Displays errors using Blueprint `Callout` components. Each input field shows relevant messages below them if validation fails or if there's an issue during submission.
5. **Theme Toggle**:
   - The `ThemeToggle` component allows users to switch between light and dark themes from the registration page.
6. **BlueprintJS Styling**:
   - Uses Blueprint components for a cohesive UI experience:
     - `Card`: Wraps the registration form with a card layout.
     - `Button`, `FormGroup`, `InputGroup`: For structured form elements.
     - `H1`: Displays the "Register" title prominently.
   - Inline styles are applied for centering and responsiveness.

Backend Requirements

The backend developer should implement an endpoint to handle user registration. Below is an outline:

1. **Registration Endpoint**:
   - **Endpoint**: `/api/auth/register`
   - **Method**: `POST`
   - **Expected Payload**:

     ```
     {
       "username": "string",
       "uid": "integer" // Unique identifier for the user
     }
     ```

   - **Response**:
     - **Success**: Return a success message or user object.

       ```
       {
         "message": "Registration successful",
         "user": {
           "id": "integer",
           "username": "string",
           "uid": "integer"
         }
       }
       ```

     - **Error**: Return an error message for various validation failures (e.g., "Username already exists", "Password must be at least 8 characters long").
2. **User Creation Logic**:
   - Ensure that the backend checks for existing usernames to avoid duplicates.
   - Store the UID and other user details in the database upon successful registration.

This integration will create a smooth registration experience in the application.

AdminDashboard.jsx

Overview

The `AdminDashboard` component serves as the main interface for administrators. It includes navigation elements, a loading spinner for data fetching, and clickable cards that lead to various management sections.

Key Functionalities

- **User Authentication:** Utilizes the `logout` function from the `AuthContext` to allow admins to log out of their session.
- **Loading State:** Displays a loading spinner while simulating data fetching to enhance user experience.
- **Navigation Cards:** Clickable cards represent different administrative functionalities, like user management, system settings, and analytics.

Component Structure

1. **Imports:**
   - React and necessary hooks (`useEffect`, `useState`) for state management and lifecycle methods.
   - UI components from BlueprintJS for styling and structure.
   - A `ThemeToggle` component for switching themes.
   - `useAuth` context for handling authentication-related actions.
2. **State Management:**
   - `loading`: A state variable to manage the loading status of the dashboard.
3. **Effect Hook:**
   - Simulates an asynchronous operation (data fetching) by using a timer that sets the loading state to false after 2 seconds.
4. **Rendering Logic:**
   - If `loading` is true, a spinner is displayed.
   - Once loading is complete, a navbar is rendered with:
     - **Title:** Displays "Admin Dashboard".
     - **Theme Toggle:** Allows the user to switch between dark and light themes.
     - **Logout Button:** Triggers the `logout` function when clicked.
   - Below the navbar, three clickable cards represent sections of the dashboard:
     - **User Management**
     - **System Settings**
     - **Analytics**

Proposed Backend API Endpoints

To fully implement the functionalities within this component, the following backend endpoints may be needed:

1. **Logout Endpoint:**
   - **URL:** `/api/logout/`

    - **Method:** `POST`
    - **Description:** Logs the user out of the session.
2. **User Management Endpoint:**
    - **URL:** `/api/users/`
    - **Method:** `GET`
    - **Description:** Fetches a list of users for management purposes (e.g., view, edit, delete).
3. **System Settings Endpoint:**
    - **URL:** `/api/settings/`
    - **Method:** `GET` and `PUT`
    - **Description:** Fetches and updates system-wide settings.
4. **Analytics Endpoint:**
    - **URL:** `/api/analytics/`
    - **Method:** `GET`
    - **Description:** Retrieves analytics data to display on the dashboard.

Example Usage

When the admin navigates to this dashboard:

- The component initiates loading state.
- After 2 seconds, the loading spinner is replaced with the navbar and management cards.
- Each card’s click handler can be expanded to navigate to corresponding detailed views or actions.

BusinessOwnerDashboard.jsx

Overview

The `BusinessOwnerDashboard` component is the main interface for business owners, providing them access to invoices, estimates, and client management functionalities. It integrates theme switching and user authentication, displaying relevant information upon loading.

Key Functionalities

- **User Welcome Message:** Displays a personalized greeting using the username from the authentication context.
- **Loading and Error Handling:** Manages loading states and displays error alerts if data fetching fails.
- **Navigation:** Allows business owners to navigate to different sections of the application through clickable cards.

Component Structure

1. **Imports:**
   - React and necessary hooks (`useEffect`, `useState`) for managing component lifecycle and state.
   - BlueprintJS components for user interface elements.
   - `useNavigate` from React Router for navigation between routes.
   - A `ThemeToggle` component for switching themes.
   - `useAuth` context for managing user authentication state.
2. **State Management:**
   - `loading`: A boolean state variable to indicate whether data is being fetched.
   - `error`: A state variable to capture and display error messages.
3. **Effect Hook:**
   - `fetchData`: A function simulating an API call using a timeout to mimic loading data. It sets `loading` to false after 2 seconds.
   - In case of an error, it sets the `error` state and stops the loading.
4. **Rendering Logic:**
   - Displays a spinner while data is loading.
   - If an error occurs, it displays an alert with the error message.
   - Once loading is complete, it renders the dashboard layout:
     - A welcome header that displays the user's name and includes theme toggling and logout functionality.
     - Three clickable cards representing:
       - Invoices
       - Estimates
       - Clients
   - Clicking on a card triggers navigation to the respective page.

Proposed Backend API Endpoints

To facilitate the functionalities of this component, consider implementing the following backend API endpoints:

1. **User Information Endpoint:**
   - **URL:** `/api/user/`
   - **Method:** `GET`
   - **Description:** Retrieves the current authenticated user's information, including the username.
2. **Logout Endpoint:**
   - **URL:** `/api/logout/`
   - **Method:** `POST`
   - **Description:** Logs the user out of the session.
3. **Invoices Endpoint:**
   - **URL:** `/api/invoices/`
   - **Method:** `GET`
   - **Description:** Fetches a list of invoices associated with the business owner.
4. **Estimates Endpoint:**
   - **URL:** `/api/estimates/`
   - **Method:** `GET`
   - **Description:** Retrieves estimates created by the business owner.
5. **Clients Endpoint:**
   - **URL:** `/api/clients/`
   - **Method:** `GET`
   - **Description:** Fetches a list of clients associated with the business owner.

Example Usage

Upon accessing the dashboard:

- The component fetches user data and displays a loading spinner for 2 seconds.
- After loading, it shows a welcome message, theme toggle, and logout button.
- Clicking any of the cards will navigate the user to the corresponding route (invoices, estimates, or clients).

EstimateForm.jsx

Overview

The `EstimateForm` component allows users to create estimates by adding items, their quantities, units, and prices. It dynamically updates and displays a summary of the estimates in a table format.

Key Functionalities

- **Input Form:** Users can enter the item name, quantity, unit of measurement, and price per unit.
- **Estimate List:** Displays a summary table of all added estimates, showing item details and calculated totals.

Component Structure

1. **Imports:**
   - React and hooks (`useState`) for managing state.
   - BlueprintJS components for building the user interface, including buttons and input fields.
   - Optional CSS for custom styling.
2. **State Management:**
   - `itemName`: Stores the name of the item being estimated.
   - `quantity`: Stores the quantity of the item.
   - `unit`: Stores the selected unit of measurement.
   - `price`: Stores the price per unit of the item.
   - `estimates`: An array that holds all the estimates added by the user.
3. **Unit Options:**
   - An array `unitOptions` contains predefined unit types that the user can select from.
4. **Form Submission:**
   - The `handleAddEstimate` function is triggered on form submission:
     - It prevents the default form submission behavior.
     - Checks if all required fields are filled.
     - Creates a new estimate object with a unique ID, item details, and total price calculated from quantity and price.
     - Updates the `estimates` state with the new estimate and resets the form fields.
5. **Rendering Logic:**
   - Renders a form with input fields for item name, quantity, unit, and price.
   - Includes a button to add the item to the estimate.
   - If there are any estimates, it displays a summary table showing:
     - Item Name
     - Quantity
     - Unit
     - Price per Unit
     - Total

Proposed Backend API Endpoints

To support the functionality of this component, the following backend API endpoints may be useful:

1. **Create Estimate Endpoint:**
   - **URL:** `/api/estimates/`
   - **Method:** `POST`
   - **Payload:**

     ```
     json
     Copy code
     {
       "itemName": "String",
       "quantity": "Number",
       "unit": "String",
       "price": "Number",
       "total": "Number"
     }
     ```

   - **Description:** Saves a new estimate to the database.
2. **Fetch Estimates Endpoint:**
   - **URL:** `/api/estimates/`
   - **Method:** `GET`
   - **Description:** Retrieves a list of estimates associated with the user.
3. **Delete Estimate Endpoint:**
   - **URL:** `/api/estimates/{id}/`
   - **Method:** `DELETE`
   - **Description:** Deletes an estimate by its unique identifier.

Example Usage

When a user interacts with the `EstimateForm` component:

- They enter the item name, quantity, unit, and price, then submit the form.
- The form adds the estimate to the list and clears the input fields.
- The summary table updates dynamically, displaying all entered estimates with calculated totals.

EstimateTable.jsx

Overview

The `EstimateTable` component displays a table of estimates, allowing users to view, delete, and calculate total estimates. It also includes functionalities for generating a PDF and sharing estimates via WhatsApp.

Key Functionalities

- **Estimates Table:** Displays a list of estimates with item details and actions.
- **View Estimate Details:** Opens a modal to show details of a selected estimate.
- **Delete Estimates:** Allows users to remove estimates from the list.
- **Total Calculation:** Calculates and displays the total amount of all estimates.
- **PDF Generation and Sharing:** Provides options to generate a PDF of the estimates and share them via WhatsApp.

Component Structure

1. **Imports:**
   - React and hooks (`useState`) for managing component state.
   - BlueprintJS components for creating the table and UI elements, including buttons and modals.
   - Custom components `GeneratePDF` and `ShareToWhatsApp` for generating PDFs and sharing functionality.
2. **State Management:**
   - `estimates`: An array holding the list of estimates.
   - `selectedEstimate`: Stores the currently selected estimate to view its details.
   - `isModalOpen`: Boolean state to control the visibility of the modal.
3. **Estimate Management:**
   - **handleDelete:** Function to remove an estimate from the list based on its ID.
   - **handleView:** Function to set the selected estimate and open the modal to view details.
   - **calculateTotalEstimate:** Computes the total of all estimates in the list.
4. **Rendering Logic:**
   - Renders a table with columns for item name, quantity, unit, price per unit, total, and action buttons (view and delete).
   - Displays a message if no estimates are available using `NonIdealState`.
   - Shows the total estimate amount at the bottom of the table along with buttons for PDF generation and sharing.
   - The modal displays detailed information for the selected estimate when the view button is clicked.

Proposed Backend API Endpoints

To support the functionality of this component, the following backend API endpoints may be necessary:

1. **Fetch Estimates Endpoint:**
   - **URL:** `/api/estimates/`

    - **Method:** `GET`
    - **Description:** Retrieves a list of estimates associated with the user.
2. **Delete Estimate Endpoint:**
    - **URL:** `/api/estimates/{id}/`
    - **Method:** `DELETE`
    - **Description:** Deletes an estimate by its unique identifier.
3. **Generate PDF Endpoint:**
    - **URL:** `/api/estimates/generate-pdf/`
    - **Method:** `POST`
    - **Payload:**

      ```
      json
      Copy code
      {
        "estimates": [
          {
            "itemName": "String",
            "quantity": "Number",
            "unit": "String",
            "price": "Number",
            "total": "Number"
          }
        ]
      }
      ```

    - **Description:** Generates a PDF document for the provided estimates.

Example Usage

When a user interacts with the `EstimateTable` component:

- Estimates are fetched and displayed in a table format.
- Users can click on the "View" button to see more details in a modal.
- The "Delete" button allows users to remove any estimate from the table.
- The total amount for all estimates is calculated and displayed.
- Users can generate a PDF of the estimates or share them via WhatsApp.

InvoiceForm.jsx

Overview

The `InvoiceForm` component allows users to input invoice item details, including item name, quantity, unit, and price. It displays a summary of the invoice items in a table format. Users can add multiple items to the invoice.

Key Functionalities

- **Input Fields:** Users can enter item details, which include item name, quantity, unit, and price per unit.
- **Invoice Summary Table:** Displays a summary of all added invoice items, including total amounts calculated based on quantity and price.
- **Dynamic Updates:** Automatically updates the invoice summary as items are added.

Component Structure

1. **Imports:**
   - React and hooks (`useState`) for managing component state.
   - BlueprintJS components for form elements and UI, including buttons and tables.
   - Custom CSS for styling.
2. **State Management:**
   - `itemName`: Holds the name of the item being added.
   - `quantity`: Holds the quantity of the item.
   - `unit`: Holds the selected unit of measurement for the item.
   - `price`: Holds the price per unit of the item.
   - `invoices`: An array to store the added invoice items.
3. **Unit Options:**
   - `unitOptions`: An array of predefined units (e.g., 'Piece', 'Kg', 'Liter', etc.) for users to select from when adding items.
4. **Adding an Invoice Item:**
   - **handleAddInvoice:** This function creates a new invoice item object with a unique ID, calculates the total based on quantity and price, and adds it to the `invoices` state. It also resets the input fields for the next entry.
5. **Rendering Logic:**
   - Renders a form with input fields for item name, quantity, unit, and price.
   - Displays a table of added invoices if there are any items in the `invoices` array.
   - The table shows columns for item name, quantity, unit, price per unit, and total.

Proposed Backend API Endpoints

To support the functionality of this component, the following backend API endpoints may be necessary:

1. **Fetch Invoices Endpoint:**
   - **URL:** `/api/invoices/`
   - **Method:** `GET`
   - **Description:** Retrieves a list of invoices associated with the user.
2. **Create Invoice Endpoint:**

- **URL:** `/api/invoices/`
- **Method:** `POST`
- **Payload:**

```
json
Copy code
{
  "invoices": [
    {
      "itemName": "String",
      "quantity": "Number",
      "unit": "String",
      "price": "Number",
      "total": "Number"
    }
  ]
}
```

- **Description:** Creates a new invoice with the provided invoice items.

Example Usage

When a user interacts with the `InvoiceForm` component:

- The user fills out the form with item details and clicks the "Add to Invoice" button.
- The item gets added to the invoice list, and the invoice summary table updates to reflect the newly added item.
- Users can continue adding items, and the table displays all entries with their respective details and totals.

InvoiceTable.jsx

Overview

The `InvoiceTable` component displays a list of invoices, allowing users to view details, delete invoices, and calculate the total amount for all invoices. It integrates functionality for generating PDF files and sharing invoices via WhatsApp.

Key Functionalities

- **Invoice Listing:** Displays a table of invoices with details such as item name, quantity, unit, price, and total.
- **Actions:** Users can view details of an invoice or delete it from the list.
- **Total Calculation:** Automatically calculates and displays the total amount for all invoices.
- **Modal for Viewing Invoice Details:** Users can view detailed information about a selected invoice in a modal.

Component Structure

1. **Imports:**
    - React and hooks (`useState`) for managing component state.
    - BlueprintJS components for tables, buttons, modals, and headers.
    - `GeneratePDF` and `ShareToWhatsApp` components for additional functionalities.
2. **State Management:**
    - `invoices`: An array to store the invoice data.
    - `selectedInvoice`: Holds the currently selected invoice for viewing details.
    - `isModalOpen`: A boolean state to manage the visibility of the modal.
3. **Handling Invoices:**
    - **handleDelete:** Removes an invoice from the list based on its ID.
    - **handleView:** Sets the selected invoice and opens the modal to view its details.
4. **Total Calculation:**
    - **calculateTotalInvoice:** A function that calculates the total of all invoices by summing their total amounts.
5. **Rendering Logic:**
    - Renders a table containing the list of invoices.
    - Each invoice has columns for item name, quantity, unit, price per unit, total, and action buttons.
    - Displays a total invoice amount below the table.
    - A modal appears when a user clicks to view an invoice’s details, showing additional information.

Proposed Backend API Endpoints

To support the functionality of this component, the following backend API endpoints may be necessary:

1. **Fetch Invoices Endpoint:**
    - **URL:** `/api/invoices/`
    - **Method:** `GET`
    - **Description:** Retrieves a list of invoices associated with the user.

2. **Delete Invoice Endpoint:**
   - **URL:** `/api/invoices/{id}/`
   - **Method:** `DELETE`
   - **Description:** Deletes a specific invoice identified by its ID.
3. **Create Invoice Endpoint (if applicable):**
   - **URL:** `/api/invoices/`
   - **Method:** `POST`
   - **Payload:**

     ```
     json
     Copy code
     {
       "itemName": "String",
       "quantity": "Number",
       "unit": "String",
       "price": "Number",
       "total": "Number"
     }
     ```

   - **Description:** Creates a new invoice with the provided details.

Example Usage

When a user interacts with the `InvoiceTable` component:

- The user sees a table of invoices, each with action buttons to view or delete.
- Clicking "View" opens a modal displaying detailed information about the selected invoice.
- Clicking "Delete" removes the invoice from the list and updates the total invoice amount accordingly.
- At the bottom, the user can generate a PDF of the invoices or share them via WhatsApp.

GeneratePDF.jsx

Overview

The `GeneratePDF` component allows users to generate and download a PDF document based on the provided data. This component utilizes the `jspdf` and `jspdf-autotable` libraries to create a formatted PDF that includes a table with relevant information.

Key Functionalities

- **PDF Generation:** Converts the given data into a structured PDF document with a title and a table format.
- **Empty State Handling:** Displays a message when there is no data available for generating a PDF.

Component Structure

1. **Imports:**
   - React for building the component.
   - BlueprintJS components (`Button`, `EmptyState`, `EmptyStateIcon`, `EmptyStateBody`) for UI elements.
   - `jsPDF` and `jspdf-autotable` for PDF creation and table formatting.
2. **Props:**
   - `data`: An array of objects that contains the data to be included in the PDF.
   - `type`: A string that indicates the type of document (e.g., "invoice" or "estimate").
3. **PDF Generation Logic:**
   - The `generatePDF` function is called when the user clicks the button to generate the PDF.
   - A new `jsPDF` instance is created, and a title is added to the document.
   - The `data` prop is mapped to a format suitable for the PDF table.
   - The `autoTable` method is used to create a table within the PDF, with headers for item details.
   - Finally, the PDF is saved with a filename based on the `type` prop.
4. **Empty State Handling:**
   - If the `data` array is empty or undefined, the component displays an empty state message with an icon, indicating that no data is available for PDF generation.

Example Usage

When a user interacts with the `GeneratePDF` component:

- If data is provided, clicking the "Generate PDF" button will create and download a PDF document containing the specified data.
- If no data is available, a message will be displayed stating that no data is available for generating the PDF.

Example of Data Structure

The `data` prop should be an array of objects, each with the following structure:

```
json
Copy code
[
  {
    "itemName": "String",
    "quantity": Number,
    "unit": "String",
    "price": Number,
    "total": Number
  }
]
```

Example:

```
json
Copy code
[
  {
    "itemName": "Sample Item 1",
    "quantity": 10,
    "unit": "Piece",
    "price": 50,
    "total": 500
  },
  {
    "itemName": "Sample Item 2",
    "quantity": 5,
    "unit": "Kg",
    "price": 75,
    "total": 375
  }
]
```

Proposed Backend API Endpoints

While this component primarily focuses on the frontend functionality, the following backend API could be useful for generating documents:

1. **Fetch Data for PDF Generation:**
   - **URL:** `/api/invoices/` or `/api/estimates/`
   - **Method:** `GET`
   - **Description:** Retrieves the data necessary for generating the PDF.

Overview

The `ShareToWhatsApp` component allows users to share a generated PDF document via WhatsApp. It includes a modal for entering a client's phone number and validates the input before creating a PDF and generating a WhatsApp sharing link.

Key Functionalities

- **Phone Number Validation:** Ensures the entered phone number is in a valid format.
- **PDF Generation:** Creates a PDF document using the provided data when the user decides to share the document.
- **WhatsApp Sharing:** Generates a link to share the PDF document via WhatsApp.

Component Structure

1. **Imports:**
    - React for building the component.
    - BlueprintJS components (`Button`, `Modal`, `FormGroup`, `InputGroup`, `Toaster`, `Position`, `Callout`) for UI elements.
    - `jsPDF` and `jspdf-autotable` for PDF creation and table formatting.
    - WhatsApp icon for visual representation.
2. **State Variables:**
    - `isModalOpen`: Boolean state to control the visibility of the modal.
    - `phoneNumber`: String state to store the user's input for the phone number.
    - `errorMessage`: String state to store any error messages that occur during validation.
3. **Methods:**
    - **`validatePhoneNumber(number)`**: Validates the entered phone number against a regular expression to ensure it is in the correct format.
    - **`generatePDF()`**: Creates a PDF document that includes a table of the invoice data. It also calculates the total amount and saves the PDF file.
    - **`shareOnWhatsApp()`**: Validates the phone number and checks if there is data available to share. If valid, it generates the PDF and constructs a WhatsApp sharing link, which opens in a new tab.
4. **Render Logic:**
    - A button to open the modal for WhatsApp sharing.
    - The modal includes:
        - An input field for the client's WhatsApp number.
        - Error messages for validation failures or absence of data.
        - Send and Cancel buttons to trigger sharing or close the modal.
    - A `Toaster` component to display success messages when the document is shared.

Example Usage

When a user interacts with the `ShareToWhatsApp` component:

- Clicking the "Share via WhatsApp" button opens a modal.
- The user can enter a WhatsApp number and click "Send".

- If the input is valid and data is present, the PDF is generated, and a sharing link is created and opened in WhatsApp.

Example of Data Structure

The `data` prop should be an array of objects, each with the following structure:

```
json
Copy code
[
  {
    "itemName": "String",
    "quantity": Number,
    "unit": "String",
    "price": Number,
    "total": Number
  }
]
```

Example:

```
json
Copy code
[
  {
    "itemName": "Sample Item 1",
    "quantity": 10,
    "unit": "Piece",
    "price": 50,
    "total": 500
  },
  {
    "itemName": "Sample Item 2",
    "quantity": 5,
    "unit": "Kg",
    "price": 75,
    "total": 375
  }
]
```

Proposed Backend API Endpoints

While this component focuses on frontend functionality, it may require a backend service for the following purposes:

1. **Send Document via WhatsApp:**
    - **URL:** `/api/share-via-whatsapp/`
    - **Method:** `POST`
    - **Description:** Optionally, this endpoint could handle the logic of sending documents via WhatsApp if further integration is desired.