Legal Document Management Interface

Project Overview

Develop a responsive Legal Document Management Interface using ReactJS that allows users to upload, view, and manage legal documents (PDFs). The application should dynamically adjust its layout based on screen size and provide intuitive modals for uploading and viewing documents with mock data extractions. Optionally, implement a NodeJS backend to handle file uploads and return mock extraction data.

Technical Stack

- **Frontend:**
  - ReactJS
  - CSS Flexbox for responsive design
  - React Modal or similar library for modals
  - PDF.js or react-pdf for PDF previews
- **Backend (Optional):**
  - NodeJS with Express

UI/UX Design

General Layout

- **Page 1: Document Dashboard**
  - **Header:** Icon in the top-left corner.
  - **Content Area:** Nine boxes labeled "Legal Document 1" through "Legal Document 9".
  - **Responsive Design:** Use Flexbox to ensure dynamic positioning based on screen size.
  - Display of upload date and file name upon successful upload of a document.
- **Page 2: Upload Modal**
  - Popup modal triggered by clicking on a document box.
  - File upload interface accepting only PDF files.
- **Page 3: Document Details Modal**
  - Split-screen modal:
    - **Left:** PDF preview.
    - **Right:** Mock extractions with page numbers and "Go To Page" buttons.
  - Header displaying the name of the selected document.
  - Close (X) button at the top-right corner.

Functional Specifications

Page 1: Document Dashboard

- **Icon:**
    - Place an icon in the top-left corner (e.g., a folder or document icon).
- **Document Boxes:**
    - Create nine boxes labeled from "Legal Document 1" to "Legal Document 9".
    - Each box should be a flex item, adjusting based on screen size.
    - **On Click:**
        - **If no document uploaded:** Open Upload Modal (Page 2).
        - **If document exists:** Open Document Details Modal (Page 3).
- **Displaying Uploaded Information:**
    - After a successful upload, each box should display:
        - File Name
        - Upload Date

Page 2: Upload Modal

- **Trigger:** Clicking on a document box without an uploaded file.
- **Features:**
    - File input accepting only PDF files.
    - Submit button to upload the file.
    - Validation to ensure only PDFs are uploaded.
    - Upon successful upload:
        - Capture and store the file name.
        - Capture and store the upload date.
        - Display the above information on the respective document box.
        - Close the modal.
- **Error Handling:**
    - Display error messages for invalid file types or upload failures.

Page 3: Document Details Modal

- **Trigger:** Clicking on a document box with an uploaded file.
- **Layout:**
    - **Header:** Title displaying the selected document's name and an X button to close the modal.
    - **Left Panel:** PDF preview of the uploaded document.
    - **Right Panel:**
        - List of mock extractions (e.g., Extraction 1, Extraction 2).
        - Each extraction displays the page number it appears on.
        - "Go To Page" button next to each extraction to navigate to the respective page in the PDF preview.

- **Features:**
    - PDF preview should be scrollable and support page navigation.
    - "Go To Page" buttons should smoothly scroll the PDF preview to the specified page.
- **Mock Extractions:**
    - Generate random page numbers for each extraction using a randomizer.
    - Example:
        - Extraction 1 - Page 3
        - Extraction 2 - Page 7

Additional Notes

- **Responsive Design:**
    - Utilize CSS Flexbox to ensure that the document boxes rearrange gracefully on different screen sizes.
    - Test the application on various devices to ensure usability.
- **State Management:**
    - Use React's `useState` and `useEffect` hooks for managing component states and side effects.
    - Consider using Context API or Redux if the state becomes complex.
- **File Handling:**
    - Since there's no persistent storage, uploaded files and their metadata will only persist during the session.
    - Utilize Local Storage to retain data across page reloads if necessary.
- **PDF Handling:**
    - Use libraries like `react-pdf` for rendering PDF previews within the application.
    - Ensure that the "Go To Page" functionality interacts correctly with the PDF viewer to navigate to the specified page.
- **Error Handling & Validation:**
    - Implement robust error handling for file uploads and API interactions.
    - Provide user-friendly error messages and feedback.
- **Code Quality:**
    - Maintain clean and readable code with proper commenting.
    - Follow best practices for React and NodeJS development.
- **Testing:**
    - Perform manual testing to ensure all functionalities work as expected.
    - Optionally, write unit tests for critical components and functionalities.

Nice to Have

The following features are considered optional and should be implemented **only if time permits**, prioritized in the order listed below:

1. **Add Comment Blocks for Backend Integration**
    - **Code Documentation:** In areas of the frontend code where backend interactions are anticipated, include detailed comment blocks to outline future integration points.
        - **API Endpoint References:** Specify the intended API endpoints, request methods (e.g., GET, POST), and expected data formats.
        - **Data Flow Descriptions:** Explain how data will flow between the frontend and backend, including any necessary transformations or validations.
        - **Technology Stack Suggestions:** Recommend technologies or libraries that could be used for backend integration, such as **Axios** for HTTP requests or **Redux** for state management.
        - **Error Handling Notes:** Outline strategies for handling potential errors or exceptions that may arise during backend communication.
2. **Implement a Simple NodeJS App**
    - **Backend Development:** Develop the optional NodeJS backend to handle file uploads and provide mock extraction data, enhancing the application's functionality.
        - **Express Server Setup:** Create an Express.js server with necessary middleware for handling requests and responses.
        - **File Upload Handling:** Use **Multer** middleware to process multipart/form-data for PDF uploads, ensuring only valid PDF files are accepted.
        - **API Endpoints:**
            - `POST /upload`: Handle incoming PDF file uploads, store them temporarily, and respond with relevant metadata.
            - `GET /extractions/:documentId`: Return mock extraction data, including randomized page numbers for each extraction.
        - **CORS Configuration:** Implement **CORS** middleware to allow secure communication between the frontend and backend during local development.
        - **Data Management:** Since persistent storage is not required, manage uploaded files and extraction data in-memory or use simple file storage solutions.
        - **Integration Points:** Ensure the frontend can communicate seamlessly with the backend by configuring appropriate API calls and handling responses correctly.
3. **Implement Unit Tests**
    - **Testing Frameworks:** Incorporate unit testing to ensure code reliability and maintainability.
        - **Frontend Testing:** Use **Jest** and **React Testing Library** to write tests for React components, ensuring they render correctly and handle user interactions as expected.
        - **Backend Testing:** Utilize **Jest** or **Mocha** with **Supertest** to write tests for backend API endpoints, verifying correct responses and error handling.

        - **Test Coverage:** Aim for comprehensive test coverage of critical functionalities, including:
            - File upload validation and handling.
            - Modal opening and closing behaviors.
            - PDF preview rendering and page navigation.
            - Data extraction display and "Go To Page" functionality.
        - **Continuous Integration:** Optionally set up a CI pipeline to run tests automatically on code commits, ensuring ongoing code quality.
4. **Improve the Design**
    - **Enhanced UI/UX:** Elevate the user interface beyond the basic layout to create a more engaging and intuitive experience.
        - **Styling Frameworks:** Utilize CSS frameworks such as **Tailwind CSS**, **Material-UI**, or **Styled Components** to achieve a modern and consistent design.
        - **Responsive Enhancements:** Ensure that all components not only adjust dynamically but also maintain aesthetic appeal across various devices and screen sizes.
        - **Accessibility:** Incorporate accessibility best practices, including proper ARIA labels, keyboard navigation support, and sufficient color contrast.
        - **Animations and Transitions:** Add subtle animations or transitions to improve user interactions, such as button hover effects or modal entry animations.
        - **Theming:** Implement light and dark mode themes to enhance user personalization and comfort.