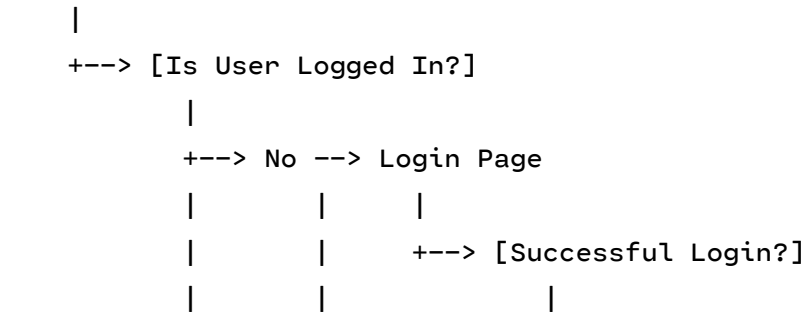
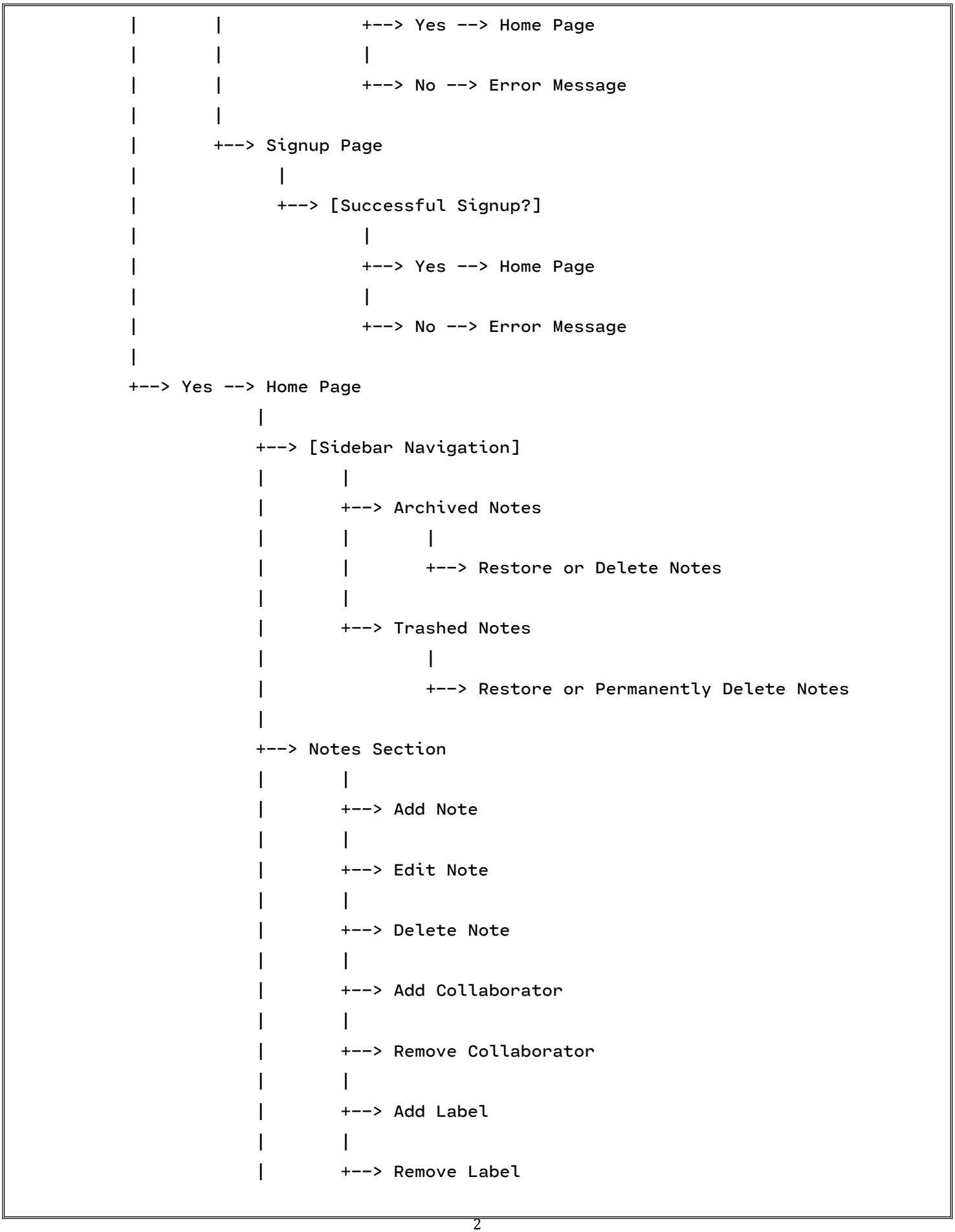
User Flow Diagram

Flowchart Breakdown:

1. **Landing Page**
   - User arrives at the landing page.
   - If the user is logged in, they are redirected to the Home page.
   - If the user is not logged in, they are directed to the Login page.
2. **Login Page**
   - User enters credentials.
   - On successful login, user is redirected to the Home page.
   - Option to navigate to the Signup page if the user doesn't have an account.
3. **Signup Page**
   - User enters registration details.
   - On successful signup, user is logged in and redirected to the Home page.
4. **Home Page**
   - Displays all user notes.
   - Options to add, edit, delete notes.
   - Options to add, remove collaborators.
   - Options to add, remove labels.
5. **Sidebar Navigation**
   - Options to view archived notes, trashed notes.
   - Option to create and delete labels.
6. **Archive and Trash Pages**
   - Display archived or trashed notes respectively.
   - Options to restore or permanently delete notes.

Diagram




| | +--> Yes --> Home Page
| | |
| | +--> No --> Error Message
| |
| +--> Signup Page
| |
| +--> [Successful Signup?]
| |
| +--> Yes --> Home Page
| |
| +--> No --> Error Message
|
+--> Yes --> Home Page
|
+--> [Sidebar Navigation]
| |
| +--> Archived Notes
| | |
| | +--> Restore or Delete Notes
| |
| +--> Trashed Notes
| |
| +--> Restore or Permanently Delete Notes
|
+--> Notes Section
| |
| +--> Add Note
| |
| +--> Edit Note
| |
| +--> Delete Note
| |
| +--> Add Collaborator
| |
| +--> Remove Collaborator
| |
| +--> Add Label
| |
| +--> Remove Label

```
                    |
                    +--> Create Label
                            |
                            +--> Delete Label
```

Explanation:

- **Landing Page:** The entry point of your application. Checks if the user is logged in and redirects accordingly.
- **Login Page:** Allows the user to log in. On successful login, redirects to the Home page.
- **Signup Page:** Allows the user to sign up. On successful signup, logs the user in and redirects to the Home page.
- **Home Page:** The main page where the user can see all their notes and perform various actions like adding, editing, and deleting notes, managing collaborators and labels.
- **Sidebar Navigation:** Contains links to view archived notes, trashed notes, and manage labels.
- **Archived Notes:** Displays notes that have been archived. Allows restoring or deleting notes.
- **Trashed Notes:** Displays notes that have been trashed. Allows restoring or permanently deleting notes.
- **Notes Section:** Main area on the Home page to manage notes.
- **Create Label:** Allows the user to create and delete labels.

Component Design

Planning:

1. **Break down the UI into reusable components.**
2. **Define the props and state each component needs.**
3. **Plan the hierarchy and nesting of components.**

Detailed Approach:

1. **Atomic Design Principles:**
    - **Atoms:** Basic building blocks of the UI (e.g., buttons, input fields).
    - **Molecules:** Combinations of atoms (e.g., form groups, note cards).
    - **Organisms:** Groups of molecules forming distinct sections (e.g., sidebars, headers).

- **Templates:** Layouts combining organisms (e.g., home page layout).
- **Pages:** Specific instances of templates (e.g., home page, login page).

2. **Component Hierarchy:**

Here's a hierarchy chart for your application:

```
App
|
+-- LandingPage
|   |
|   +-- Header
|   +-- Footer
|
+-- LoginPage
|   |
|   +-- LoginForm
|
+-- SignupPage
|   |
|   +-- SignupForm
|
+-- HomePage
    |
    +-- Header
    +-- Sidebar
    |   |
    |   +-- NavigationLink
    |   +-- CreateLabelForm
    |
    +-- NotesSection
        |
        +-- NoteForm
        +-- NoteList
            |
            +-- NoteCard
                |
                +-- CollaboratorList
                +-- LabelList
    +-- ArchivePage
        |
```

```
        +-- NoteList
            |
            +-- NoteCard
    +-- TrashPage
        |
        +-- NoteList
            |
            +-- NoteCard
```

Components Breakdown

1. Atoms:

- **Button:** A basic button component.
- **InputField:** A basic input field component.

```jsx
// Button.js
import React from 'react';

const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);

export default Button;

// InputField.js
import React from 'react';

const InputField = ({ value, onChange, placeholder }) => (
  <input value={value} onChange={onChange} placeholder={placeholder} />
);

export default InputField;
```

2. Molecules:

- **LoginForm:** Combination of input fields and a button.
- **SignupForm:** Similar to LoginForm but for signing up.
- **NoteCard:** Displays a single note.
- **NoteForm:** Form to add/edit a note.

```jsx
// LoginForm.js
import React, { useState } from 'react';
import InputField from './InputField';
import Button from './Button';

const LoginForm = ({ onSubmit }) => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    onSubmit({ email, password });
  };

  return (
    <form onSubmit={handleSubmit}>
      <InputField value={email} onChange={e => setEmail(e.target.value)} placeholder="E
mail" />
      <InputField value={password} onChange={e => setPassword(e.target.value)} placehol
der="Password" />
      <Button>Login</Button>
    </form>
  );
};

export default LoginForm;

// NoteCard.js
import React from 'react';

const NoteCard = ({ note, onDelete }) => (
  <div>
    <h3>{note.title}</h3>
    <p>{note.content}</p>
    <button onClick={() => onDelete(note.id)}>Delete</button>
  </div>
);

export default NoteCard;
```

3. Organisms:

- **Header:** Contains navigation links.
- **Sidebar:** Contains links to archived and trashed notes, create label form.
- **NotesSection:** Contains NoteForm and NoteList.

```
// Header.js
import React from 'react';

const Header = () => (
  <header>
    <h1>Noting App</h1>
    <nav>
      <a href="/">Home</a>
      <a href="/archive">Archive</a>
      <a href="/trash">Trash</a>
    </nav>
  </header>
);

export default Header;

// Sidebar.js
import React from 'react';
import CreateLabelForm from './CreateLabelForm';

const Sidebar = () => (
  <aside>
    <nav>
      <a href="/archive">Archived Notes</a>
      <a href="/trash">Trashed Notes</a>
    </nav>
    <CreateLabelForm />
  </aside>
);

export default Sidebar;

// NotesSection.js
import React from 'react';
```

```
import NoteForm from './NoteForm';
import NoteList from './NoteList';

const NotesSection = () => (
  <section>
    <NoteForm />
    <NoteList />
  </section>
);

export default NotesSection;
```

4. Templates:

- **HomePage:** Combines Header, Sidebar, and NotesSection.
- **ArchivePage:** Combines Header, Sidebar, and NoteList for archived notes.
- **TrashPage:** Combines Header, Sidebar, and NoteList for trashed notes.

```
// HomePage.js
import React from 'react';
import Header from './Header';
import Sidebar from './Sidebar';
import NotesSection from './NotesSection';

const HomePage = () => (
  <div>
    <Header />
    <div className="main-content">
      <Sidebar />
      <NotesSection />
    </div>
  </div>
);

export default HomePage;

// ArchivePage.js
import React from 'react';
import Header from './Header';
import Sidebar from './Sidebar';
import NoteList from './NoteList';
```

```jsx
const ArchivePage = () => (
  <div>
    <Header />
    <div className="main-content">
      <Sidebar />
      <NoteList />
    </div>
  </div>
);

export default ArchivePage;

// TrashPage.js
import React from 'react';
import Header from './Header';
import Sidebar from './Sidebar';
import NoteList from './NoteList';

const TrashPage = () => (
  <div>
    <Header />
    <div className="main-content">
      <Sidebar />
      <NoteList />
    </div>
  </div>
);

export default TrashPage;
```

Final Steps:

1. **Create Components:** Implement each component based on the breakdown above.
2. **Integrate Components:** Combine components to build the main pages.
3. **Style Components:** Use CSS or a styling library to style your components.
4. **Test Components:** Ensure each component works as expected.

State Management Design Using Context API

Planning:

1. **Determine the state management solution:**
    - Use Context API for managing global states such as authentication, user data, and notes.
2. **Identify global and local states:**
    - **Global States:** Authentication state, user data, notes data, labels data, collaborators data.
    - **Local States:** Form inputs, modal visibility, UI-specific states like toggling sidebar, note edit state.
3. **Context Providers:**
    - Create context providers for global states and wrap the application with these providers.

Visual Design

Contexts Overview

1. **AuthContext:**
    - Manages authentication state, user data, and handles login, logout, and token refresh.
2. **NotesContext:**
    - Manages notes data, including fetching, adding, updating, deleting, and state of individual notes (archived, trashed).
3. **LabelsContext:**
    - Manages labels data, including creating, updating, and deleting labels.
4. **CollaboratorsContext:**
    - Manages collaborators data, including adding and removing collaborators from notes.

Context Providers and State Structure

```
App
|
+-- AuthProvider
|   |
|   +-- Provides: isAuthenticated, user, login, logout, refreshAccessToken
|
+-- NotesProvider
|   |
|   +-- Provides: notes, fetchNotes, addNote, updateNote, deleteNote, archiveNote, tras
hNote
```

```
|
+-- LabelsProvider
|   |
|   +-- Provides: labels, fetchLabels, addLabel, updateLabel, deleteLabel
|
+-- CollaboratorsProvider
    |
    +-- Provides: collaborators, addCollaborator, removeCollaborator
```

Detailed State Management Plan

1. **AuthContext:**
   - **State Variables:** isAuthenticated, user, accessToken, refreshToken.
   - **Actions:** login, logout, refreshAccessToken.
   - **Global States Managed:**
     - Authentication status.
     - User information.
     - Access token and refresh token management.
2. **NotesContext:**
   - **State Variables:** notes, isLoading, error.
   - **Actions:** fetchNotes, addNote, updateNote, deleteNote, archiveNote, trashNote.
   - **Global States Managed:**
     - List of notes.
     - Note operations (CRUD and state changes like archive and trash).
3. **LabelsContext:**
   - **State Variables:** labels, isLoading, error.
   - **Actions:** fetchLabels, addLabel, updateLabel, deleteLabel.
   - **Global States Managed:**
     - List of labels.
     - Label operations (CRUD).
4. **CollaboratorsContext:**
   - **State Variables:** collaborators, isLoading, error.
   - **Actions:** addCollaborator, removeCollaborator.
   - **Global States Managed:**
     - List of collaborators for notes.
     - Collaborator operations (add and remove).

Local State Management

Component Specific State:

1. **Form Inputs**: Managed within the respective forms (e.g., NoteForm, LoginForm).
2. **Modal Visibility**: Managed within the components that handle modals (e.g., state variables like isModalOpen).
3. **UI-Specific States**: Managed within the respective components (e.g., toggling sidebar, note edit state).

Context Providers Hierarchy

```
<App>
  <AuthProvider>
    <NotesProvider>
      <LabelsProvider>
        <CollaboratorsProvider>
          <Router>
            <LandingPage />
            <LoginPage />
            <SignupPage />
            <HomePage>
              <Header />
              <Sidebar />
              <NotesSection>
                <NoteForm />
                <NoteList />
              </NotesSection>
            </HomePage>
            <ArchivePage>
              <Header />
              <Sidebar />
              <NoteList />
            </ArchivePage>
            <TrashPage>
              <Header />
              <Sidebar />
              <NoteList />
            </TrashPage>
          </Router>
        </CollaboratorsProvider>
```

```
      </LabelsProvider>
    </NotesProvider>
  </AuthProvider>
</App>
```

Steps to Implement State Management

1. **Create Contexts:**
   - Create separate context files for AuthContext, NotesContext, LabelsContext, and CollaboratorsContext.
2. **Set Up Providers:**
   - Implement provider components that manage the state and provide actions to update the state.
3. **Wrap Application with Providers:**
   - In the App component, wrap the entire application with the context providers to make the global state available throughout the app.
4. **Use Context in Components:**
   - Use the context in the components to access and manipulate the global state.

Design and Plan Using React Query for Your Noting Application

Overview

Conceptual Overview

1. **QueryClient**: Centralized cache and data management for queries and mutations.
2. **Query**: Function to fetch notes, labels, and collaborators, manage caching, loading, and error states.
3. **Mutation**: Function to modify notes, labels, and collaborators on the server and update the cache optimistically.
4. **Query Keys**: Unique identifiers for queries to differentiate between different data requests.
5. **Invalidations**: Mechanism to refetch data based on specific conditions (e.g., after adding or deleting a note).

Visual Diagram

React Query Data Flow for Your Noting Application

```
Noting Application
  |
```

```
+-- QueryClientProvider
|      |
|      +-- QueryClient
|            |
|            +-- QueryCache
|            +-- MutationCache
|
+-- Components
       |
       +-- Custom Hooks (useQuery, useMutation)
              |
              +-- Query Keys (notes, labels, collaborators)
              +-- Queries (Fetch notes, labels, collaborators)
              +-- Mutations (Add, update, delete notes, labels, collaborators)
```

Detailed Flow Explanation

1. **QueryClientProvider:**
   - Initializes the QueryClient, which manages caching, data fetching, and mutations across your noting application.
2. **QueryClient:**
   - Manages QueryCache and MutationCache.
   - Caches query results to optimize performance and reduce unnecessary network requests.
3. **Components:**
   - Utilize custom hooks (useQuery and useMutation) provided by React Query to interact with data.
4. **Custom Hooks:**
   - **useQuery:** Fetches notes, labels, and collaborators using specific query keys.
     - Manages loading and error states transparently.
     - Automatically updates the cache with fetched data.
   - **useMutation:** Performs mutations such as adding, updating, and deleting notes, labels, and collaborators.
     - Supports optimistic updates for a responsive user interface.
     - Handles server responses and updates the cache accordingly.
5. **Query Keys:**

- Define query keys for notes, labels, and collaborators to differentiate between different data requests.
- Facilitate efficient caching and data invalidation strategies.

Data Fetching and Mutations in Your Application

- **Fetch Data:**
  - Use useQuery hooks to fetch notes, labels, and collaborators.
  - React Query manages caching and automatically refreshes data based on invalidation policies (e.g., after a note is added or deleted).
- **Mutations:**
  - Utilize useMutation hooks to add, update, and delete notes, labels, and collaborators.
  - Benefit from optimistic updates to provide a smooth user experience while awaiting server responses.

Benefits for Your Noting Application

- **Simplified Data Management:** Reduce boilerplate code for data fetching and state management.
- **Efficient Caching:** Optimize application performance by caching data and minimizing network requests.
- **Responsive User Interface:** Enhance user experience with optimistic updates and seamless data synchronization.

Next Steps

1. **Implementation:** Integrate React Query into your noting application by setting up QueryClientProvider and implementing useQuery and useMutation hooks for notes, labels, and collaborators.
2. **Testing:** Validate data fetching and mutation functionalities across various scenarios to ensure robust performance and reliability.
3. **Refinement:** Fine-tune caching strategies and invalidation policies based on specific use cases and application requirements.

Authentication System Design and Plan

Overview

1. **JWT Authentication:** Use JSON Web Tokens (JWTs) for secure authentication.
2. **Token Management:** Handle access tokens and refresh tokens for authentication and session management.

3. **Secure Storage:** Store tokens securely, considering options like browser memory or HTTP-only cookies.
4. **Protected Routes:** Define routes that require authentication to access.

Visual Representation

Authentication System Flow

```
Noting Application
  |
  +-- User Interface
  |       |
  |       +-- LandingPage (Redirect to LoginPage if not authenticated)
  |       +-- LoginPage
  |       +-- SignupPage
  |       +-- HomePage (Protected Route)
  |             |
  |             +-- Header (Display user info and logout)
  |             +-- Sidebar (Navigation links)
  |             +-- NotesSection (Display notes and manage CRUD operations)
  |
  +-- Authentication Flow
          |
          +-- Login (POST /api/login)
          |       |
          |       +-- Validate credentials
          |       +-- Issue Access Token (JWT) and Refresh Token
          |       +-- Store Access Token
          |       +-- Redirect to HomePage
          |
          +-- Signup (POST /api/signup)
          |       |
          |       +-- Create new user account
          |       +-- Issue Access Token (JWT) and Refresh Token
          |       +-- Store Access Token
          |       +-- Redirect to HomePage
          |
          +-- Token Refresh (POST /api/refresh-token)
          |       |
          |       +-- Check validity of refresh token
          |       +-- Issue new Access Token
```

```
|      +-- Store new Access Token
|
+-- Logout (POST /api/logout)
       |
       +-- Remove tokens from storage
       +-- Redirect to LoginPage
```

Detailed Flow Explanation

1. **User Interface:**
   - **LandingPage:** Initial landing page redirects to LoginPage if not authenticated.
   - **LoginPage:** Allows users to log in using email/password.
   - **SignupPage:** Enables new users to sign up for an account.
2. **Protected Routes:**
   - **HomePage:** Requires authentication to access. Displays user-specific content like notes and allows CRUD operations.
3. **Authentication Flow:**
   - **Login:**
     - Validates user credentials.
     - Issues an access token (JWT) and a refresh token.
     - Stores the access token and redirects authenticated users to HomePage.
   - **Signup:**
     - Creates a new user account.
     - Issues an access token (JWT) and a refresh token upon successful signup.
     - Stores the access token and redirects the user to HomePage.
   - **Token Refresh:**
     - Checks the validity of the refresh token.
     - Issues a new access token if the refresh token is valid.
     - Stores the new access token for continued authenticated sessions.
   - **Logout:**
     - Removes stored tokens (access token and refresh token).
     - Redirects the user to the LoginPage for reauthentication.

Next Steps

1. **Implementation:** Implement the authentication endpoints (login, signup, refresh token, logout) on your backend using Node.js, Express, and MongoDB.
2. **Integrate Frontend:** Implement authentication flows in your React frontend using Axios or Fetch API to communicate with the backend endpoints.
3. **Secure Token Storage:** Decide whether to store tokens in memory or secure HTTP-only cookies based on your application's security requirements.
4. **Testing and Validation:** Test authentication flows thoroughly to ensure security, reliability, and a smooth user experience.

Error Handling Design and Plan Using React Query

Overview

1. **Centralized Error Handling:** Implement a centralized mechanism to handle errors globally across the application, leveraging React Query's error handling capabilities.
2. **HTTP Status Codes:** Handle specific HTTP status codes (401, 400, 404, 500, 409) with React Query's built-in error handling and custom error messages.
3. **User Feedback:** Provide clear and meaningful error messages to users using toast notifications or modals.
4. **Logging and Monitoring:** Log errors for debugging purposes and monitor them to identify recurring issues.

Visual Representation

Error Handling Flow with React Query

```
Noting Application
  |
  +-- User Interface
  |      |
  |      +-- Components (NotesSection, LoginPage, SignupPage, etc.)
  |            |
  |            +-- ErrorBoundary (Catch errors in components)
  |
  +-- React Query Integration
  |      |
  |      +-- QueryClientProvider (Setup React Query)
```

```
|      |
|      +-- Custom Hooks (useQuery, useMutation)
|            |
|            +-- Queries (Fetch notes, labels, collaborators)
|            +-- Mutations (Add, update, delete notes, labels, collaborators)
|
+-- Error Handling
       |
       +-- React Query Error Handling
       |      |
       |      +-- useQuery Error Handling
       |      +-- useMutation Error Handling
       |      +-- Display Error Message (Toast or Modal)
       |
       +-- ErrorBoundary Component (Local error handling)
              |
              +-- Catch and Display Component-Specific Errors
```

Detailed Flow Explanation

1. **User Interface:**
   - Components like NotesSection, LoginPage, SignupPage, etc., where errors might occur during user interactions.
2. **React Query Integration:**
   - **QueryClientProvider:** Setup React Query's QueryClientProvider to manage data fetching, caching, and mutations across the application.
   - **Custom Hooks (useQuery, useMutation):**
     - Use React Query's hooks (useQuery for fetching data and useMutation for mutations) in components to interact with backend APIs.
     - These hooks manage loading states, cache invalidation, and automatic retries.
3. **React Query Error Handling:**
   - **Global Error Handling:**
     - React Query provides built-in mechanisms to handle common HTTP errors (like 401, 400, 404, 500, 409) through error states in useQuery and useMutation.
     - Display appropriate error messages using toast notifications or modals within the components.

4. **ErrorBoundary Component:**
   - **Local Error Handling:**
     - Implement error boundaries around components to catch JavaScript errors and React Query errors that occur within them.
     - Display component-specific error messages or fallback UI to prevent crashes and improve user experience.

Common Error Scenarios

- **401 Unauthorized:** React Query handles token expiration or invalid tokens by automatically triggering a retry with a refreshed token if configured.
- **400 Bad Request:** Display user-friendly messages for invalid input or parameters sent to backend APIs.
- **404 Not Found:** Inform users when requested resources (e.g., notes, labels) are not found using React Query's error handling.
- **500 Internal Server Error:** Notify users of unexpected server issues and guide them on actions to take, such as retrying or contacting support.
- **409 Conflict:** Handle conflicts (e.g., duplicate entries) gracefully using React Query's mutation error handling and provide resolution steps to users.

Next Steps

1. **Implementation:** Integrate React Query into your noting application and configure error handling mechanisms using useQuery and useMutation.
2. **Error Messages:** Define and display clear error messages for each error scenario using toast notifications or modals within your components.
3. **Testing:** Test error handling scenarios extensively to ensure robustness and reliability under different conditions, such as network failures or server errors.
4. **Feedback and Improvement:** Monitor error logs, gather user feedback, and continuously refine error handling to enhance user satisfaction and application reliability.

```
noting-app/
├── public/
│   └── index.html
├── src/
│   ├── assets/
│   │   └── images/
```

```
|   ├── components/
|   |   ├── Auth/
|   |   |   ├── LoginPage.jsx
|   |   |   ├── SignupPage.jsx
|   |   |   └── AuthForms.jsx
|   |   ├── Notes/
|   |   |   ├── NotesSection.jsx
|   |   |   └── NoteCard.jsx
|   |   ├── Labels/
|   |   |   ├── LabelsSection.jsx
|   |   |   └── LabelItem.jsx
|   |   ├── Collaborators/
|   |   |   ├── CollaboratorsSection.jsx
|   |   |   └── CollaboratorItem.jsx
|   |   ├── Sidebar.jsx
|   |   └── ErrorBoundary.jsx
|   ├── hooks/
|   |   ├── useAuth.js
|   |   ├── useNotes.js
|   |   ├── useLabels.js
|   |   ├── useCollaborators.js
|   |   └── useErrorBoundary.js
|   ├── services/
|   |   ├── apiService.js
|   |   ├── authService.js
|   |   └── noteService.js
|   ├── pages/
|   |   ├── HomePage.jsx
|   |   └── LandingPage.jsx
|   ├── App.jsx
|   ├── index.js
|   ├── setupTests.js
|   └── theme/
|       └── styles.css
├── .env
├── .gitignore
├── package.json
└── README.md
```

Directory Structure Breakdown

- **public/**: Contains the index.html file and other static assets.
- **src/**: Main source directory for the application.
  - **assets/**: Holds images and other static assets used in the application.
  - **components/**: Reusable UI components used throughout the application.
    - **Auth/**: Components related to authentication (login, signup).
    - **Notes/**: Components for managing notes (NotesSection, NoteCard).
    - **Labels/**: Components for managing labels (LabelsSection, LabelItem).
    - **Collaborators/**: Components for managing collaborators (CollaboratorsSection, CollaboratorItem).
    - **Sidebar.jsx**: Component for navigation and displaying sidebar links.
    - **ErrorBoundary.jsx**: Component to catch and handle errors in components.
  - **hooks/**: Custom hooks used across the application.
    - Hooks for authentication (useAuth.js), managing notes (useNotes.js), labels (useLabels.js), collaborators (useCollaborators.js), and error boundaries (useErrorBoundary.js).
  - **services/**: Contains utility functions and API services.
    - **apiService.js**: Service for making HTTP requests using Axios or Fetch API.
    - **authService.js**: Service for handling authentication logic (login, signup, token management).
    - **noteService.js**: Service for managing CRUD operations related to notes.
  - **pages/**: React components representing different pages/routes of the application.
    - **HomePage.jsx**: Component for the main home page displaying notes, labels, and collaborators.
    - **LandingPage.jsx**: Initial landing page of the application.
  - **App.jsx**: Root component that sets up React Router and wraps the application with QueryClientProvider for React Query integration.
  - **index.js**: Entry point of the application where React is rendered into the DOM.
  - **setupTests.js**: Configuration file for setting up tests with Jest and Enzyme (if using).

    - **theme/**: Contains CSS styles or theme files used across the application (styles.css).
- **.env**: Environment configuration file for storing environment variables.
- **.gitignore**: Specifies files and directories that should be ignored by Git.
- **package.json**: Manifest file for Node.js projects, listing dependencies and scripts.
- **README.md**: Documentation file that provides information about the project, setup instructions, and usage details.

Next Steps

1. **Organize Components**: Ensure components are organized logically based on their functionality (auth, notes, labels, etc.).
2. **Implement Services**: Implement API services (apiService, authService, noteService) to interact with backend APIs using Axios or Fetch API.
3. **Integrate React Query**: Set up React Query (QueryClientProvider) in App.jsx and use useQuery and useMutation hooks in components for data fetching and mutations.
4. **Style and Testing**: Style components using CSS or a chosen styling solution (like styled-components) and set up testing using Jest and Enzyme (if preferred).