**Frontend Development with React.js - Project Documentation**

1. **Introduction:**

**Project Title: Cookbook**

**Project Description:**

The **Cookbook Project** aims to create a comprehensive and accessible collection of recipes that cater to a wide variety of tastes, dietary preferences, and cooking skill levels. The cookbook will serve as a practical guide for both novice and experienced cooks, offering recipes that range from simple everyday meals to more complex, gourmet dishes.

**Team Members:**

• Team Leader: PAVITHRA A

• Team Member: ABINAYA A

• Team Member: DHARSHINI B

• Team Member: MANOJ B

• Team Member: NAVEEN

**Project Overview**

**Purpose**

The purpose of the **Cookbook** is to provide an easy-to-use, digital platform where users can discover, organize, and prepare a wide variety of recipes. It aims to inspire cooking creativity, offer personalized meal recommendations, and make cooking more convenient by providing step-by-step instructions, nutritional information, and tools like meal planning and grocery lists. The app also fosters community engagement through recipe sharing and feedback, making cooking accessible and enjoyable for all skill levels.

**Features**

1. **Recipe Search and Filters**:

* Search by ingredients, cuisine, meal type, dietary preferences, and cooking time.

1. **Personalized Recommendations**:

* Suggested recipes based on user preferences, past activity, and dietary restrictions.

1. **Step-by-Step Instructions**:

* Clear, easy-to-follow instructions with accompanying images or videos.

1. **Nutritional Information**:

* Detailed nutritional facts for each recipe (calories, macronutrients, etc.).

1. **Meal Planning**:

* Customizable meal plans with the ability to schedule recipes for the week or month.

1. **Automated Grocery Lists**:

* Generate shopping lists based on selected recipes, with the option to check off items.

1. **Favorites and Collections**:

* Save favorite recipes and organize them into personal collections for quick access.

**Architecture**

**Component Structure**

1. **App Component**

* **Role**: The central, overarching component that controls the state of the entire application and handles routing between different sections. It connects the various features of the app (e.g., recipe search, meal planning, user accounts) and ensures that each part of the app works together smoothly.
* **Interaction**: This component interacts with all other components by passing down state, props, and routing information to render the appropriate page content.

**2. Header Component**

* **Role**: This component acts as the navigation and utility hub for the user. It includes the app's logo, the search functionality, and the user's profile information.
* **Interaction**: The Header is always visible and interacts with components like SearchBar for quick searches and UserProfileMenu for account management. It allows navigation to different sections of the app.

**3. Footer Component**

* **Role**: The Footer provides basic navigation links and app-related information like terms of service, privacy policies, and social media links.
* **Interaction**: It appears on all pages but interacts minimally with other components.

**4. Navigation Component**

* **Role**: The Navigation component provides access to the app's core sections, such as the home screen, recipe lists, meal planner, and favorites. It enhances user experience by offering a clear, organized way to explore the app.
* **Interaction**: It interacts with the MainContent component by determining which view is displayed based on user selections.

**5. SearchBar Component**

* **Role**: The primary tool for finding recipes within the app. The SearchBar allows users to input keywords and apply filters such as dietary preferences or meal types to narrow their results.
* **Interaction**: It interacts with the RecipeList component by providing search results and refined filters to display relevant recipes.

**6. MainContent Component**

* **Role**: This dynamic component displays the main page content based on user actions (e.g., viewing recipe details, searching for recipes, planning meals). It switches between different views like RecipeList, RecipeDetails, or MealPlanner based on the routing.
* **Interaction**: It interacts with all the core components (RecipeList, MealPlanner, FavoritesList) and updates dynamically based on navigation or search inputs.

**7. RecipeList Component**

* **Role**: This component displays a collection of recipes based on user searches or pre-set categories. It shows recipes in a compact format, with essential details such as image, name, and prep time.
* **Interaction**: The RecipeList is fed data from the SearchBar or a pre-defined category (e.g., breakfast, vegan). It passes selected recipes to the RecipeDetails component for full viewing.

**8. RecipeCard Component**

* **Role**: Each individual recipe is represented by a RecipeCard within the RecipeList. It includes an image, a title, a short description, and important metadata like prep time or ratings.
* **Interaction**: When clicked, the RecipeCard triggers the RecipeDetails view, passing the selected recipe information for full display.

**9. RecipeDetails Component**

* **Role**: The RecipeDetails component provides a detailed view of a single recipe, including ingredients, step-by-step instructions, nutrition facts, and user reviews. It’s the central hub for interacting with individual recipes.
* **Interaction**: It interacts with subcomponents like IngredientsList and Instructions. It also communicates with AddToMealPlanButton to add the recipe to the user's meal planner or RecipeReviews for user feedback.

**10. MealPlanner Component**

* **Role**: The MealPlanner helps users schedule meals for the week or month. It offers an interactive calendar where users can drag and drop recipes to different days and automatically generate shopping lists based on selected meals.
* **Interaction**: It interacts with the RecipeDetails and RecipeList components to pull recipes into the plan. It also integrates with the ShoppingList component to streamline grocery preparation.

**11. FavoritesList Component**

* **Role**: This component displays a personalized list of recipes that users have saved as favorites. It helps users quickly access their most-loved recipes without searching again.
* **Interaction**: It interacts with RecipeCard to display favorite recipes and with the RecipeDetails component when a user selects a recipe from their favorites.

**12. UserProfileMenu Component**

* **Role**: This component displays user-specific account information, allowing users to manage settings, view their uploaded recipes, and log out of the app.
* **Interaction**: It interacts with the overall App component for authentication and with various subcomponents for managing user data (favorites, preferences, etc.).

**13. ShoppingList Component**

* **Role**: The ShoppingList automatically generates a list of ingredients based on recipes added to the user's meal plan. Users can view and check off items as they shop.
* **Interaction**: It pulls data from the MealPlanner and RecipeDetails components, extracting the necessary ingredients to form a comprehensive grocery list.

**14. RecipeReviews Component**

* **Role**: Allows users to view and submit reviews and ratings for recipes. It adds a social element to the app, helping users make informed decisions based on others’ experiences.
* **Interaction**: It interacts with the RecipeDetails component, displaying user feedback for each recipe, and with ReviewForm for submitting new reviews.

**15. UserRecipes Component**

* **Role**: This section allows users to upload and manage their own personal recipes. It fosters community contributions and lets users keep a record of their own creations.
* **Interaction**: It interacts with RecipeForm for uploading new recipes and with RecipeList to display the user’s submitted recipes.

**16. AddToMealPlanButton Component**

* **Role**: A simple button that enables users to add a recipe to their meal plan with one click.
* **Interaction**: It interacts with the RecipeDetails and MealPlanner components to seamlessly integrate selected recipes into the user’s weekly or monthly plan.

**17. CookingTimer Component**

* **Role**: An embedded timer that helps users keep track of cooking times for specific recipes. It can be set directly within the recipe details.
* **Interaction**: It operates independently but interacts minimally with RecipeDetails to track cooking steps and times.

**18. Nutrition Info Component**

* **Role**: Displays a breakdown of the nutritional content of each recipe, such as calories, fats, proteins, and carbs. It helps users make healthier choices based on their dietary goals.
* **Interaction**: It interacts with the Recipe Details component by displaying the nutrition data directly alongside the recipe’s ingredients and instructions.

**State Management**

The **Cookbook Recipe App** would use **centralized state management** to ensure that data is shared efficiently across components, especially in areas like user preferences, recipe selections, meal planning, and favorites.

Two common approaches for state management in modern web apps include:

1. **React Context API** (for smaller, less complex state needs).
2. **Redux** or another external state management library (for more complex, scalable applications).

**Routing**

 **Single Page Application (SPA)**:

* The app will behave as a **single-page application** (SPA), where all content is loaded dynamically without reloading the page. This improves performance and user experience by quickly transitioning between views.

 **URL-Based Navigation**:

* The app’s navigation will be tied to different URLs for each feature (e.g., /recipes, /favorites, /meal-plan), allowing users to bookmark and share specific pages.

 **Route Matching**:

* Each route corresponds to a specific page or view in the app (e.g., homepage, recipe details, search results). Routes can be nested for better organization and modularity.

 **Dynamic Routing**:

* Some routes will be dynamic, like when viewing a specific recipe. For example, /recipes/:recipeId will display the details of a recipe based on the recipeId in the URL.

**Setup Instructions**

 **Prerequisites**: Install [Node.js](https://nodejs.org) and [Git](https://git-scm.com).

 **Clone Repository**:

bash

CopyEdit

git clone https://github.com/your-username/cookbook-recipe-app.git

cd cookbook-recipe-app

 **Install Dependencies**:

bash

CopyEdit

npm install

 **Set Environment Variables**: Create a .env file with your API keys and URLs.

 **Run the App**:

bash

CopyEdit

npm start

 **Build for Production**:

bash

CopyEdit

npm run build

 **Optional Testing**:

bash

CopyEdit

npm test

**Running the Application**

**Frontend**

To run the frontend server locally, follow these steps:

1. Navigate to the Client Directory: Open your terminal and go to the client directory of the project: 2. Install Dependencies: Install all the necessary dependencies by running:

3. Start the Development Server: After the dependencies are installed, start the development server using:

4. Access the Application: Once the server starts, open your browser and navigate to http://localhost:3000 to view the application.

**Component Documentation**

**Key Components**

 **App Component**: The root of the application, managing routing between different pages (e.g., Home, Recipes, Details) using React Router.

 **Header Component**: Provides navigation and search functionality, making it easy for users to move between major sections of the app.

 **HomePage Component**: The landing page, showcasing featured content or popular recipes to engage users immediately.

 **RecipeList Component**: Displays a list/grid of recipes, allowing users to browse or filter them.

 **RecipeCard Component**: A small UI block for each recipe, showing its image, title, and a link to detailed information.

 **RecipeDetails Component**: Shows detailed information for a specific recipe, including ingredients and cooking instructions.

 **Footer Component**: Displays links to social media, privacy policies, and other essential information at the bottom of the page.

**Reusable Components**

* **RecipeCard Component**
* **Purpose**: Displays a concise preview of a recipe, including the image, title, and a link to more detailed information. It can be reused wherever a list of recipes needs to be shown (e.g., home page, search results, favorites).
* **Why Reusable**: The same card structure is used to display recipes across multiple pages, ensuring a consistent appearance and behavior for recipe previews.

### ****Button Component****

* **Purpose**: A generic button component that can be customized with different labels, styles, and actions (e.g., submit, add to favorites, search).
* **Why Reusable**: Rather than creating multiple buttons across the app, this component can be reused with different props for labels and behaviors.

### ****Modal Component****

* **Purpose**: Displays a modal (popup) for various use cases like confirmation dialogs, viewing details, or displaying forms.
* **Why Reusable**: The same modal structure can be reused across the app for different purposes, such as confirming actions (e.g., delete recipe) or showing additional recipe details.

### ****InputField Component****

* **Purpose**: A reusable input field component that can handle various types of input (text, email, password) with validation and custom styles.
* **Why Reusable**: Input fields are needed in forms across the app (e.g., search bar, login, sign-up). This component ensures consistency in styling and functionality for all input elements.

### ****RecipeList Component****

* **Purpose**: A list/grid display of recipes that uses the **RecipeCard** component to render individual items.
* **Why Reusable**: The list can be reused to show recipes in different contexts (search results, categories, featured recipes) while maintaining consistent styling.

### ****Loader Component****

* **Purpose**: Displays a loading spinner or animation while data is being fetched or processed in the background.
* **Why Reusable**: Loading indicators are often needed in various parts of the app (e.g., loading recipes, submitting a form), and having a reusable loader ensures consistency and avoids repetition.

### ****Alert Component****

* **Purpose**: Displays alert messages such as success, error, or warning notifications (e.g., "Recipe added to favorites").
* **Why Reusable**: Alert messages are commonly needed in various scenarios, such as form validation, saving data, or handling errors. Having a reusable alert ensures a consistent design and behavior.

**State Management**

**Global State**

### ****Global State with Context API****

The **Context API** is a built-in feature in React that allows sharing state across the entire app without the need to pass props manually at every level.

You can create a context to store the **user's favorites list** or **authentication status**, and any component can access or modify this data.

* **Creating a Context for Global State**:

jsx

CopyEdit

import React, { createContext, useState, useContext } from 'react';

const GlobalStateContext = createContext();

export function GlobalStateProvider({ children }) {

const [favorites, setFavorites] = useState([]);

const [isAuthenticated, setIsAuthenticated] = useState(false);

const addFavorite = (recipe) => {

setFavorites((prevFavorites) => [...prevFavorites, recipe]);

};

const value = {

favorites,

isAuthenticated,

addFavorite,

setIsAuthenticated,

};

return (

<GlobalStateContext.Provider value={value}>

{children}

</GlobalStateContext.Provider>

);

}

export function useGlobalState() {

return useContext(GlobalStateContext);

}

* **Using Global State in Components**:

jsx

CopyEdit

import React from 'react';

import { useGlobalState } from './GlobalStateContext';

function RecipeDetails({ recipe }) {

const { addFavorite, isAuthenticated } = useGlobalState();

return (

<div>

<h2>{recipe.title}</h2>

{isAuthenticated && (

<button onClick={() => addFavorite(recipe)}>

Add to Favorites

</button>

)}

</div>

);

}

export default RecipeDetails;

* **When to use**: The **Context API** is great for managing global state in small to medium apps where state needs to be shared across different parts of the app (e.g., user info, theme settings, language preferences, favorites).

### ****2. Global State with Redux****

For larger applications with more complex state logic or when state needs to be deeply interconnected, **Redux** is a popular choice. Redux provides a **centralized store** where all the app’s state resides, and components can dispatch actions to update the state.

* **Setting Up Redux in the Cookbook App**:
  1. Install Redux and React-Redux:

bash

CopyEdit

npm install @reduxjs/toolkit react-redux

* 1. **Creating a Recipe Slice**: Redux uses slices to manage parts of the global state. Here’s an example slice for handling recipes.
  2. **Setting Up the Redux Store**: The store is the centralized place where all slices are combined, and the global state is stored.
  3. **Using Redux in Components**: Components can **dispatch actions** to update the global state or **select data** from the store using **useDispatch** and **useSelector**.
* **When to use**: Redux is best for large applications where state is complex and needs to be shared across various parts of the app, or when actions and state changes need to be logged and managed carefully (e.g., handling recipes, user authentication, favorites, cart management).

### ****Redux Toolkit for Simplified Global State****

The **Redux Toolkit** is a simpler, more efficient way of using Redux, and it reduces the boilerplate code needed to set up Redux in an app. It handles many of the common use cases automatically, like combining reducers and creating slices.

#### Benefits:

* **Simplifies setup** by providing an intuitive API for creating slices and managing asynchronous actions.
* **Better developer experience** with built-in debugging tools like Redux DevTools and Thunks for async operations.

**Local State**

### ****Managing Local State with useState****

The useState hook is used to initialize and manage local state in a component. It returns two values: the current state and a function to update that state.

In the **Cookbook Recipe App**, a local state can be used to store the value of the search input field. This state will change as the user types in the search bar.

jsx

CopyEdit

import React, { useState } from 'react';

function Header() {

const [searchQuery, setSearchQuery] = useState(''); // Initialize local state

const handleSearchChange = (e) => {

setSearchQuery(e.target.value); // Update local state on input change

};

return (

<header>

<input

type="text"

value={searchQuery}

onChange={handleSearchChange}

placeholder="Search recipes..."

/>

<p>Search term: {searchQuery}</p>

</header>

);

}

export default Header;

### ****Managing Form State Locally****

In forms, local state can be used to track multiple input fields. For example, in the recipe submission form, local state can store the title, ingredients, and instructions.

### ****Toggling UI Elements with Local State****

Local state can also be used to control the visibility of UI elements, such as a modal or dropdown menu.

jsx

CopyEdit

import React, { useState } from 'react';

function RecipeDetails() {

const [isModalOpen, setIsModalOpen] = useState(false); // Initialize local state for modal visibility

const toggleModal = () => {

setIsModalOpen((prev) => !prev); // Toggle modal visibility

};

return (

<div>

<button onClick={toggleModal}>View Recipe Details</button>

{isModalOpen && (

<div className="modal">

<h3>Recipe Details</h3>

<p>Some detailed information about the recipe.</p>

<button onClick={toggleModal}>Close</button>

</div>

)}

</div>

);

}

export default RecipeDetails;

### ****Handling Loading States Locally****

Local state can be used to indicate loading states while data is being fetched or processed.

**User Interface**

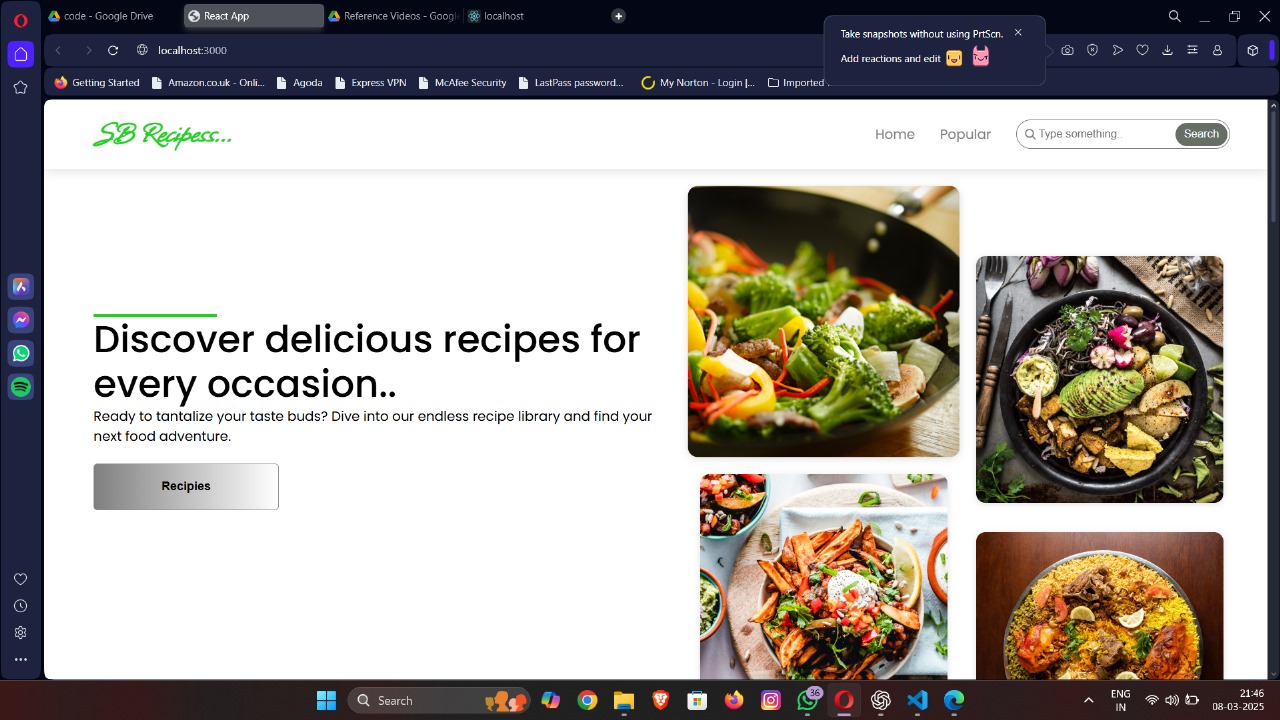
**Screenshots / GIFS**

While I can’t generate actual screenshots or GIFs, here’s a description of the user interface features and where you can add your own visuals:

 **Header**

* **Purpose**: The header provides quick access to the main features of the app, such as searching for recipes, navigating between different sections (e.g., home, favorites, profile), and displaying user-specific options (e.g., login/logout).
* **Components**:
  + **Logo**: Represents the brand or app.
  + **Search Bar**: Allows users to quickly search for recipes by name, ingredients, or keywords.
  + **Navigation Links**: Links to different sections such as Home, Favorites, My Recipes, and Profile.
  + **User Profile Icon**: Shows the user's avatar and a dropdown menu for login/logout or profile settings.

**Screenshots Example:**



 **Recipe List**

* **Purpose**: Displays a list of available recipes in a grid or list view, allowing users to quickly browse through them.
* **Components**:
  + **Recipe Card**: Each card represents a recipe and includes an image, title, short description, and possibly a rating or favorite button.
  + **Pagination or Infinite Scroll**: To handle large datasets, pagination or infinite scroll is implemented for easy navigation through multiple recipes.

 **Recipe Details Page**

* **Purpose**: Provides detailed information about a specific recipe, including ingredients, instructions, and user reviews.
* **Components**:
  + **Recipe Image**: A high-quality image of the dish.
  + **Title and Description**: The name of the recipe and a brief description or story behind it.
  + **Ingredients List**: A list of ingredients needed to make the dish.
  + **Step-by-Step Instructions**: Clear and concise instructions on how to prepare the recipe.
  + **User Reviews and Ratings**: Allows users to leave reviews or ratings for the recipe.

 **Favorites Section**

* **Purpose**: Displays the user’s favorite recipes for quick access.
* **Components**:
  + **Favorite Recipe Cards**: Similar to the recipe list, but only displays the user’s saved or favorited recipes.
  + **Remove from Favorites Button**: Allows users to remove a recipe from their favorites list.

 **Recipe Submission Form**

* **Purpose**: Allows users to submit their own recipes by filling out a form with the necessary details.
* **Components**:
  + **Input Fields for Title, Description, Ingredients, and Instructions**.
  + **Image Upload**: Option to upload an image of the dish.
  + **Submit Button**: To submit the recipe for review or to be added to the app.

 **Modals and Popups**

* **Purpose**: Used for smaller interactions like confirmation dialogs (e.g., deleting a recipe), error messages, or additional details (e.g., viewing a recipe in a popup).
* **Components**:
  + **Confirmation Dialogs**: For actions like deleting a recipe or confirming a favorite.
  + **Popup for Quick Recipe View**: A modal to view a recipe without navigating away from the current page.

**Color Scheme**

The primary color combination for the app is dark blue and white, which provides a clean and professional look, with good contrast for readability.

• Primary Color:

o Dark Blue (#0D3B66): This color is used for header backgrounds, buttons, and links to ensure a strong presence and maintain readability.

• Secondary Color:

o White (#FFFFFF): The background color and text are primarily white, creating a high-contrast look that makes the content stand out.

• Accent Colors:

o Light Blue (#1F77B4): Used for hover effects and accent elements like buttons or selected items to add vibrancy and interactivity.

o Gray (#F5F5F5): A light gray color used for secondary background areas to help the dark blue elements stand out.

Bottom of Form