CS5010: Functional Programming Project:

Recipe Ingredient, Inventory & Suggestion App

Name: Jonathan Tarun Rajasekaran

While I'm not a self-proclaimed cooking enthusiast, I've often experienced the frustration of opening my fridge only to find that one key ingredient is missing - I usually make some ham,egg and cheese sandwiches and often find that I'm missing one of the three! This recurring inconvenience inspired me to create the **Recipe Ingredient, Inventory & Suggestion App**. The application is designed to help everyday cooks manage their pantry inventory, automatically suggest recipes based on the ingredients on hand, and generate shopping lists for missing items. Built using React's functional components and Hooks—eschewing classes entirely—and leveraging Firestore for persistent storage, this project embodies a modern, modular approach to web development.

Business Requirements

- **Pantry Management:**
  Users can add, update, and delete pantry items, including details such as quantity, expiration date, and category.

- **Dynamic Recipe Suggestions:**
  The app will filter and suggest recipes that can be made with the current inventory. Users can also search and filter recipes based on cuisine, dietary preferences, or cooking time.

- **Automated Shopping List Generation:**
  When key ingredients are low or missing, the application automatically generates a shopping list for easy restocking.

- **Persistent Data Storage:**
  All data (pantry items, recipe preferences, shopping lists) is stored in Firestore, ensuring real-time updates and synchronization across devices.

- **User Interface:**
  The UI must be clean, intuitive, and responsive, supporting both desktop and mobile usage.

- **Modular and Maintainable Code:**
  The project will adhere to a functional programming paradigm with each component defined as a function (one export per file) and no global variables, ensuring a scalable and maintainable codebase.

Nouns & Verbs

Nouns:

- User
- Pantry
- Ingredient
- Recipe
- Shopping List
- Inventory
- Firestore
- ~~Preference~~
- ~~Notification~~

Verbs:

- Add
- Update
- Delete
- Suggest
- Generate
- Save
- Retrieve
- Filter
- Sync
- ~~Organize~~

Target Audience

The application is aimed at four primary user groups:

1. **Everyday Home Cooks:**
    - **Description:** Individuals who want a simple, practical tool to track pantry items and plan meals without being culinary experts.

2. **Busy Professionals:**
    - **Description:** Users who need quick, efficient meal planning and inventory management on the go.
3. **Budget-Conscious Shoppers:**
    - **Description:** Users who aim to reduce waste and optimize grocery shopping by ensuring they use what they already have.
4. **Tech-Savvy Foodies:**
    - **Description:** Individuals who appreciate modern web technology to optimize meal planning and inventory management.

Use Cases

Use Case 1: Pantry Inventory Management

- **User Story 1:**
  As a user, I want to add ingredients to my pantry so that I can track what I have available.

- **User Story 2:**
  As a user, I want to update the quantity or details of an ingredient so that my inventory stays accurate.

- **User Story 3:**
  As a user, I want to delete an ingredient from my pantry when it's no longer available so that my list remains current.

Use Case 2: Recipe Suggestions

- **User Story 1:**
  As a user, I want the app to suggest recipes based on the current ingredients in my pantry so that I can quickly decide what to cook.

- **User Story 2:**
  As a user, I want to see the required ingredients and quantities for each recipe so that I

know if I have enough supplies.

- **User Story 3:**
  As a user, I want to click a "Make Recipe" button that deducts the used quantities from my pantry so that the app helps me manage inventory automatically.

Use Case 3: Shopping List Generation

- **User Story 1:**
  As a user, I want the app to automatically generate a shopping list for ingredients that are low or missing so that I can easily plan my grocery shopping.

- **User Story 2:**
  As a user, I want to be able to mark items on the shopping list as purchased to keep my inventory up-to-date.

- **User Story 3:**
  As a user, I want the shopping list to update in real time as my pantry inventory changes so that I always know what I need.

Use Case 4: Real-Time Data Synchronization

- **User Story 1:**
  As a user, I want my pantry data, recipe suggestions, and shopping lists to be synced across all my devices so that I can access them anytime.

- **User Story 2:**
  As a user, I want any changes I make in the app to update immediately so that my inventory is always accurate.

- **User Story 3:**
  As a user, I want the app to function seamlessly in real time so that I never have to manually refresh data.

User Stories

Persona 1: Emily – The Busy Professional (Novice Cook, Novice Techie)

- **Profile:**
  Emily works long hours, rarely has time to plan meals, and often finds herself missing a key ingredient. She needs an app that is straightforward and quick to update.

- **User Stories:**

  1. *As Emily, I want to quickly add ingredients to my pantry so that I know what I have at any moment.*

  2. *As Emily, I want to see recipe suggestions that match my current inventory so that I can decide on a meal without extra shopping.*

  3. *As Emily, I want the app to generate a shopping list for missing ingredients so that I can plan my grocery trips efficiently.*

Persona 2: Daniel – The Budget-Conscious Shopper (Novice Cook, Expert Tech)

- **Profile:**
  Daniel is focused on reducing waste and saving money. Although he isn't a skilled cook, he uses technology to keep track of his ingredients and avoid unnecessary purchases.

- **User Stories:**

  1. *As Daniel, I want a detailed view of my pantry inventory so that I can track exactly how much of each ingredient I have.*

  2. *As Daniel, I want to update or delete items easily so that my pantry data remains accurate over time.*

  3. *As Daniel, I want to receive recipe suggestions that only include recipes I can actually make with the ingredients I have so that I don't waste money on extra items.*

Persona 3: Carla – The Aspiring Chef (Expert Cook, Moderate Tech)

- **Profile:**
  Carla enjoys cooking and experimenting with recipes but values organization in the kitchen. She wants to efficiently manage her pantry to support her culinary creativity.

- **User Stories:**

    1. *As Carla, I want to add recipes with detailed ingredient quantities and instructions so that I can follow them accurately.*

    2. *As Carla, I want to update my pantry after cooking so that I know exactly what has been used and what remains.*

    3. *As Carla, I want a shopping list that automatically adjusts based on my usage to ensure I have fresh ingredients on hand.*

Persona 4: Tim – The Tech-Savvy Foodie (Novice Cook, Expert Tech)

- **Profile:**
  Tim loves exploring new technologies and uses apps to optimize his daily life, including meal planning. Although he isn't an expert cook, he appreciates the efficiency and smart suggestions provided by modern apps.

- **User Stories:**

    1. *As Tim, I want the app to sync my pantry data across devices so that I can access it anywhere.*

    2. *As Tim, I want to be able to easily add, update, and remove pantry items through an intuitive interface so that my inventory is always current.*

    3. *As Tim, I want recipe suggestions that dynamically update as I add or use ingredients so that I always know what meals I can make.*

User Dimensions

- **Cooking Experience:**

    - **Novice:** Minimal cooking experience; prefers simple, straightforward interfaces.

    - **Expert:** Well-versed in cooking; may desire more detailed features and customization.

- **Technical Expertise:**
    - **Novice:** Limited familiarity with technology; needs a very intuitive, guided user experience.
    - **Expert:** Comfortable with modern apps; expects high performance and customizable options.

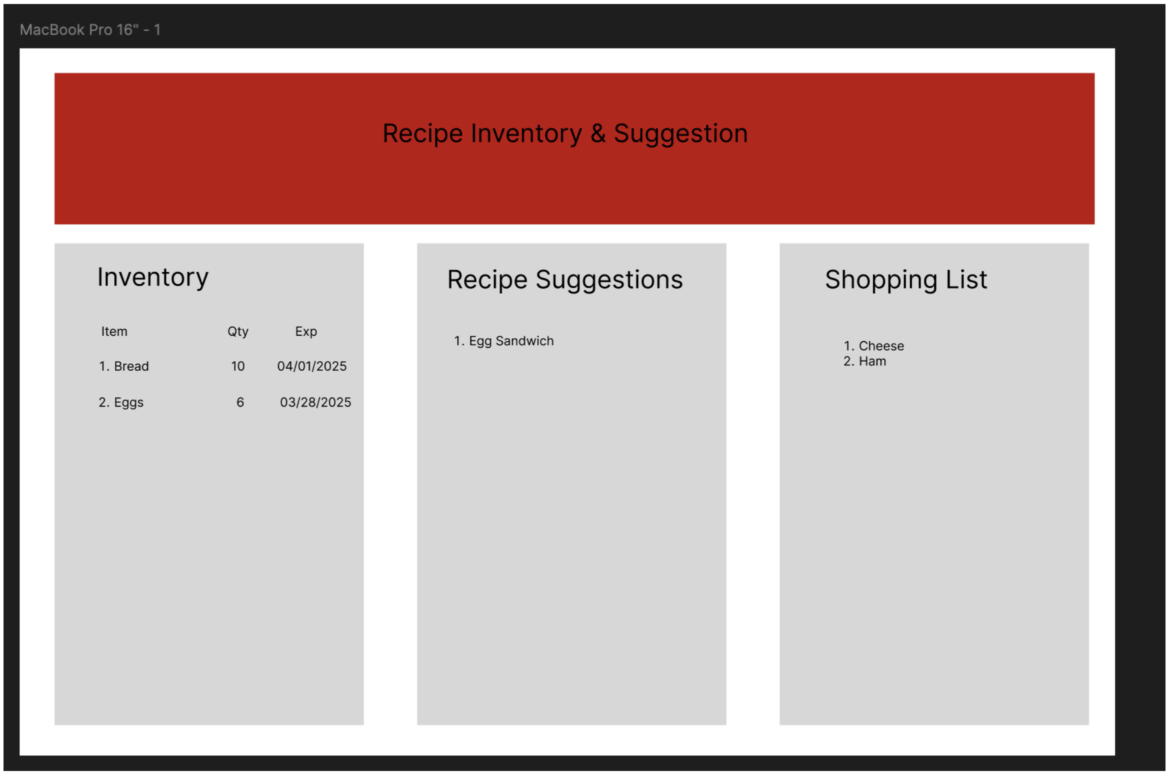
- **Time Availability / Scheduling:**
    - **Busy:** Users with limited time who need quick, streamlined interactions.
    - **Leisurely:** Users who have more time to explore features, customize settings, and engage deeply with the app.

For our personas, we select the following dimensions:

- **Cooking Experience:** We focus on users who are **novice cooks** (everyday home cooks and busy professionals).
- **Technical Expertise:** We target both **novice** (e.g., everyday home cooks) and **expert** (e.g., tech-savvy foodies) users.

Interface Mockups

- **Dashboard Screen:**
  Displays an overview of the pantry inventory, a list of recipe suggestions, and a summary of the generated shopping list.



Recipe Inventory & Suggestion

Inventory

Item	Qty	Exp
1. Bread	10	04/01/2025
2. Eggs	6	03/28/2025

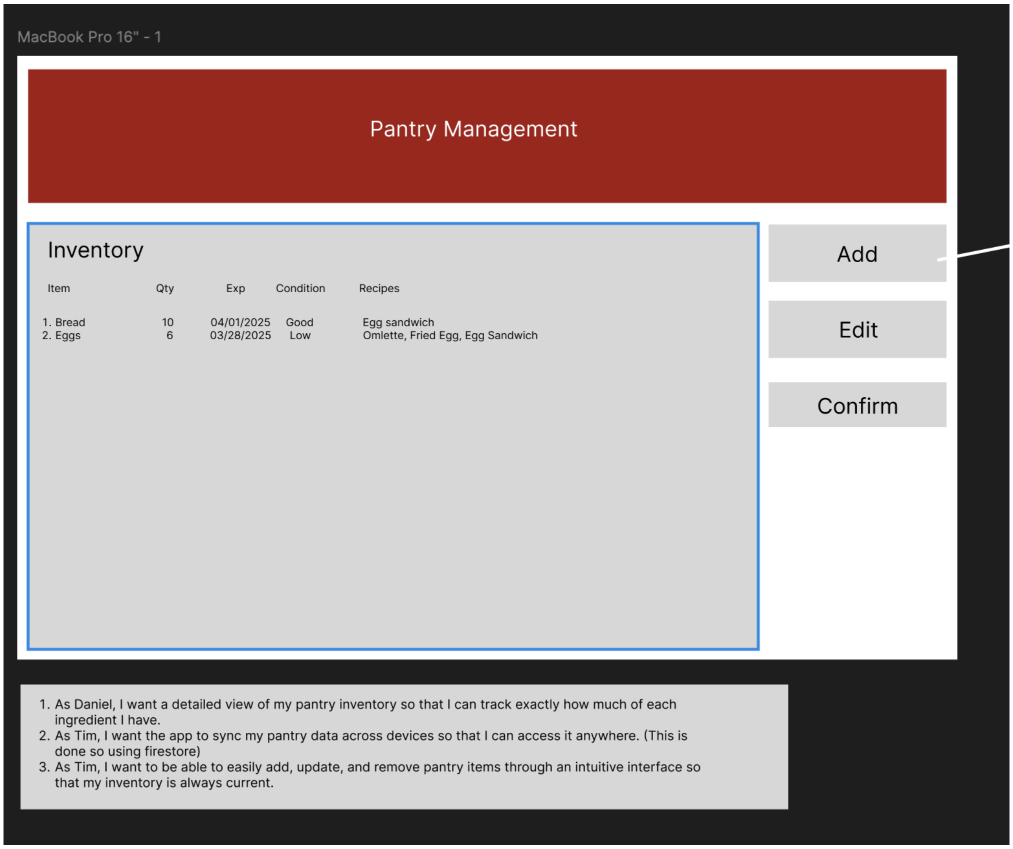
Recipe Suggestions

1. Egg Sandwich

Shopping List

1. Cheese
2. Ham

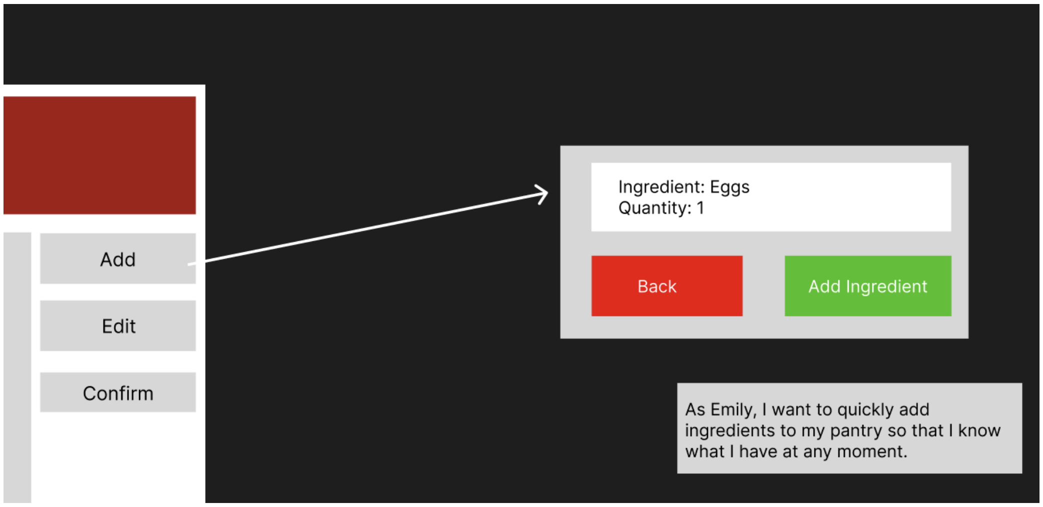
- **Pantry Management Screen:**
  A detailed interface for adding, editing, or removing ingredients. Includes fields for ingredient name, quantity, expiration date, and category. The design emphasizes ease of input and quick updates.



Inventory

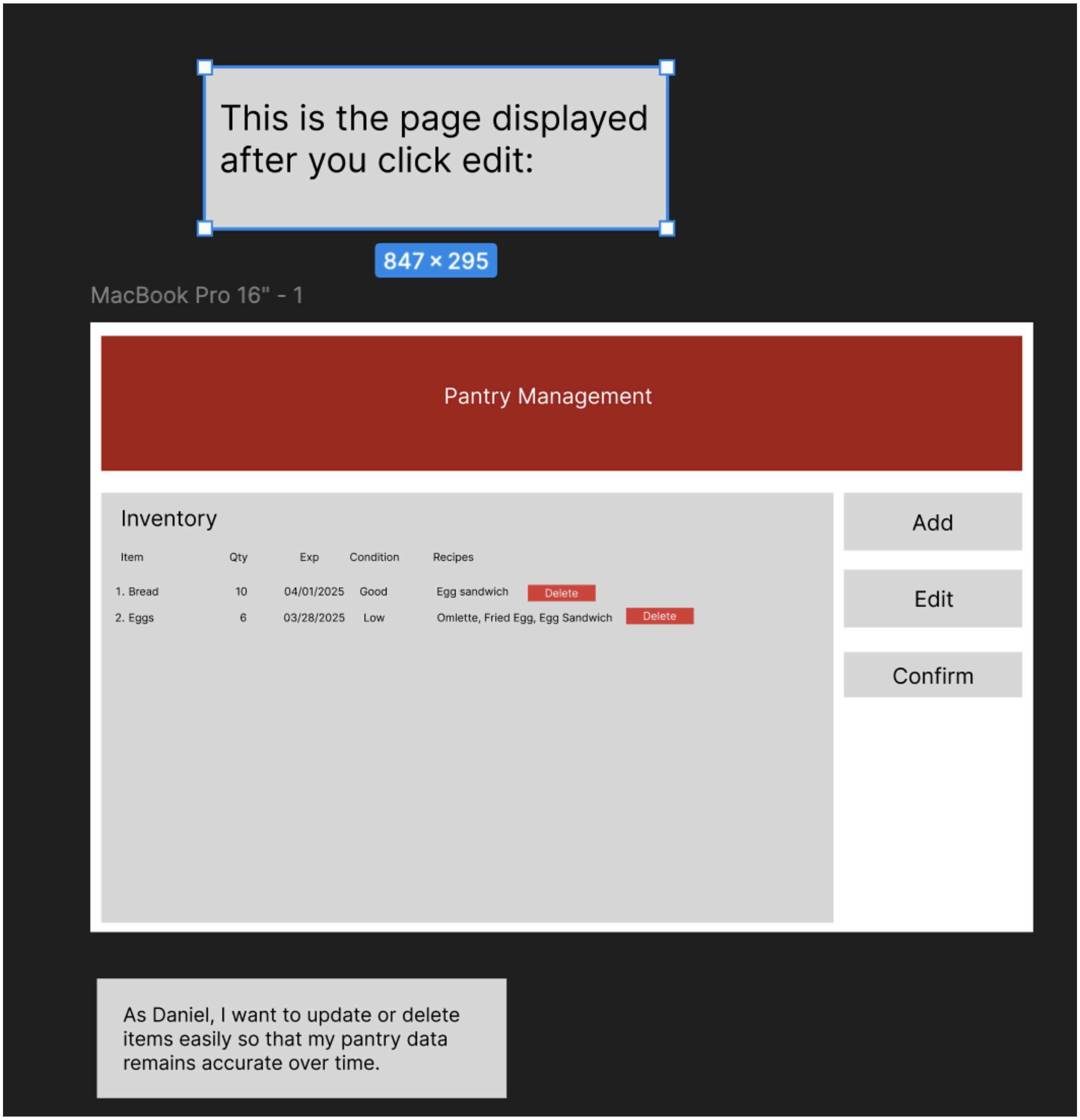
Item	Qty	Exp	Condition	Recipes
1. Bread	10	04/01/2025	Good	Egg sandwich
2. Eggs	6	03/28/2025	Low	Omlette, Fried Egg, Egg Sandwich

1. As Daniel, I want a detailed view of my pantry inventory so that I can track exactly how much of each ingredient I have.
2. As Tim, I want the app to sync my pantry data across devices so that I can access it anywhere. (This is done so using firestore)
3. As Tim, I want to be able to easily add, update, and remove pantry items through an intuitive interface so that my inventory is always current.



Ingredient: Eggs
Quantity: 1

As Emily, I want to quickly add ingredients to my pantry so that I know what I have at any moment.



This is the page displayed after you click edit:

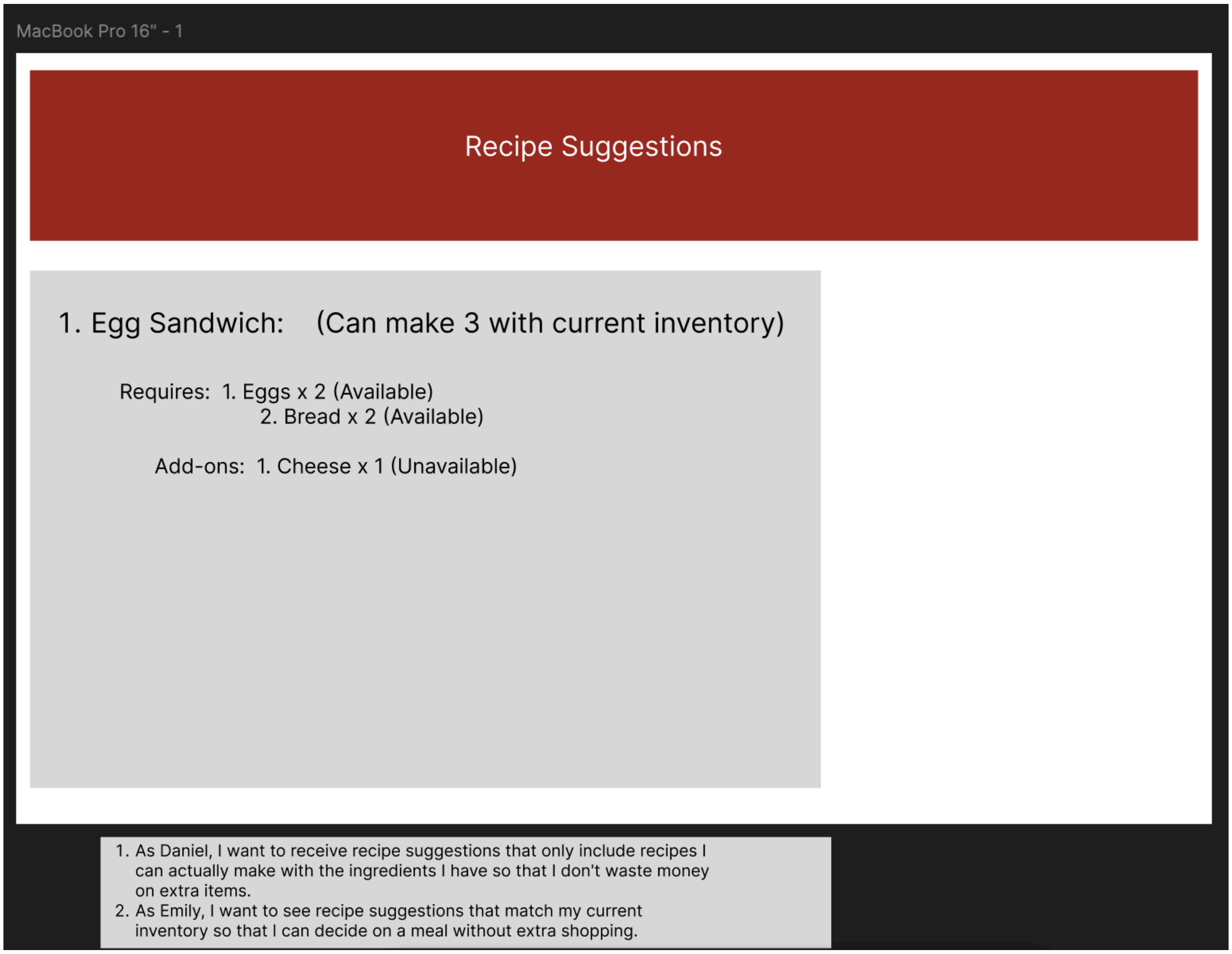
Pantry Management

Inventory

Item	Qty	Exp	Condition	Recipes	
1. Bread	10	04/01/2025	Good	Egg sandwich	Delete
2. Eggs	6	03/28/2025	Low	Omlette, Fried Egg, Egg Sandwich	Delete

As Daniel, I want to update or delete items easily so that my pantry data remains accurate over time.

- **Recipe Suggestion Screen:**
  Presents a curated list of recipes that can be prepared with the current ingredients. Users can click on a recipe to view detailed instructions, nutritional information, and preparation time.



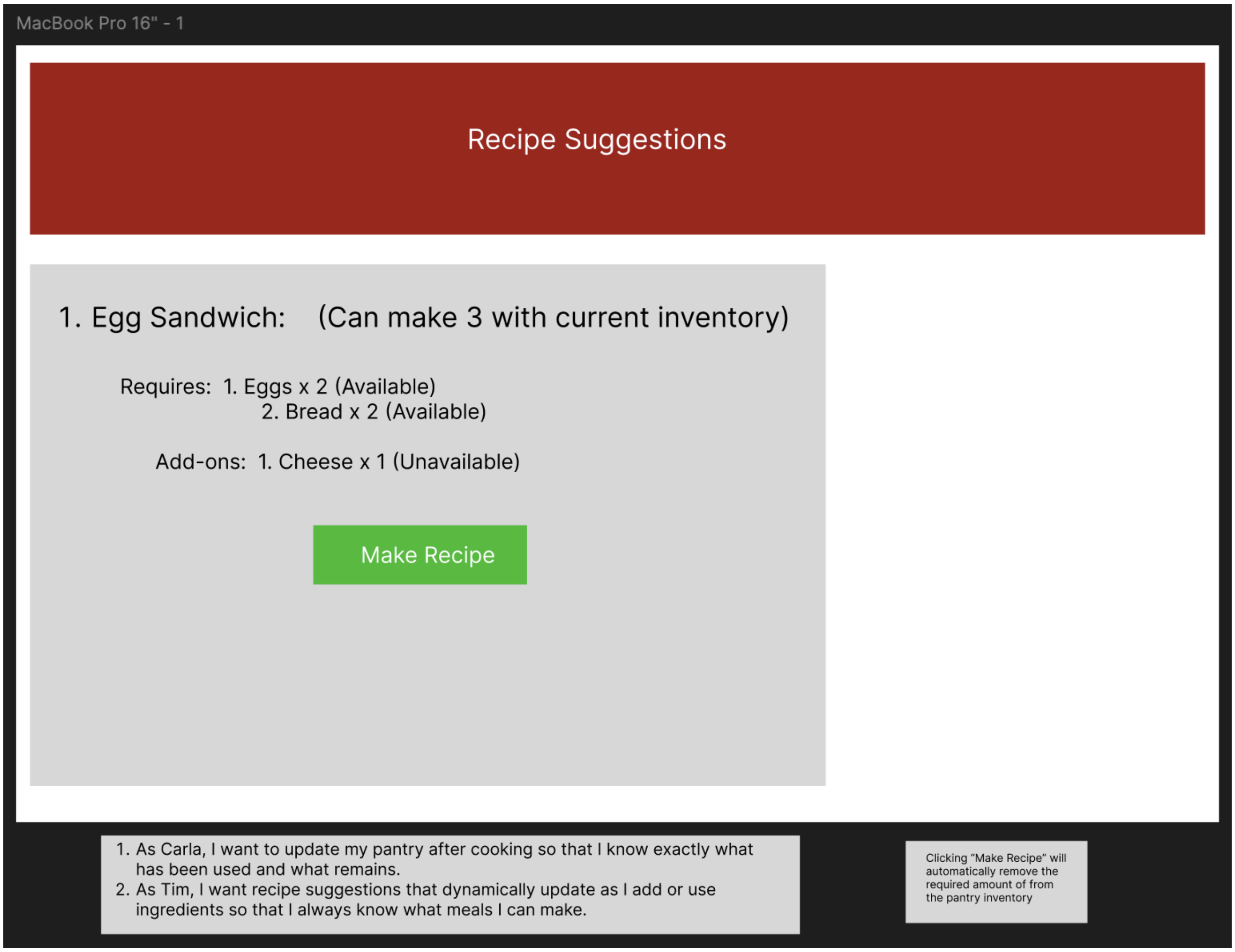
Recipe Suggestions

1. Egg Sandwich: (Can make 3 with current inventory)

Requires: 1. Eggs x 2 (Available)
2. Bread x 2 (Available)

Add-ons: 1. Cheese x 1 (Unavailable)

1. As Daniel, I want to receive recipe suggestions that only include recipes I can actually make with the ingredients I have so that I don't waste money on extra items.
2. As Emily, I want to see recipe suggestions that match my current inventory so that I can decide on a meal without extra shopping.



Recipe Suggestions

1. Egg Sandwich: (Can make 3 with current inventory)

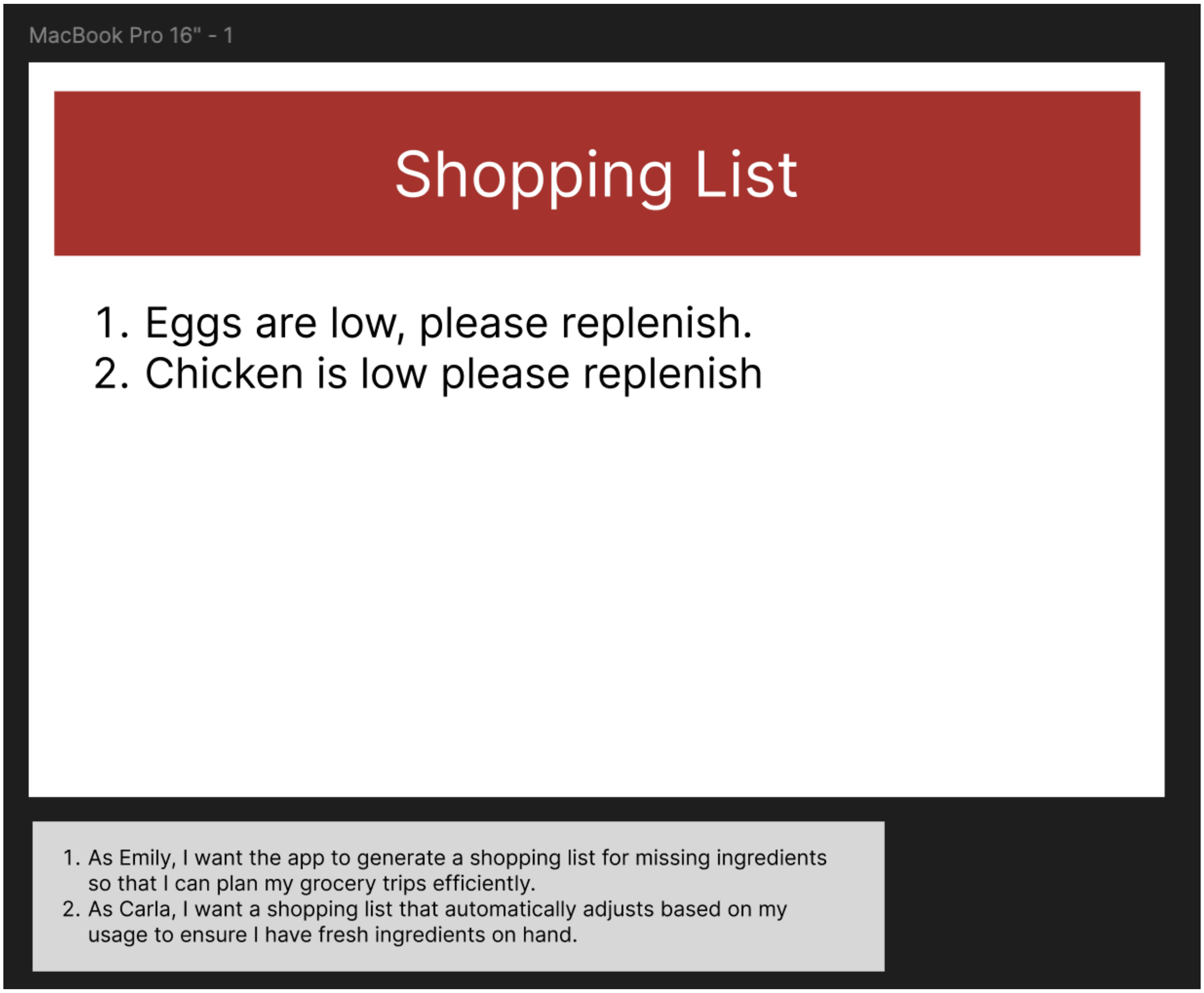
Requires: 1. Eggs x 2 (Available)
2. Bread x 2 (Available)

Add-ons: 1. Cheese x 1 (Unavailable)

1. As Carla, I want to update my pantry after cooking so that I know exactly what has been used and what remains.
2. As Tim, I want recipe suggestions that dynamically update as I add or use ingredients so that I always know what meals I can make.

Clicking "Make Recipe" will automatically remove the required amount of from the pantry inventory

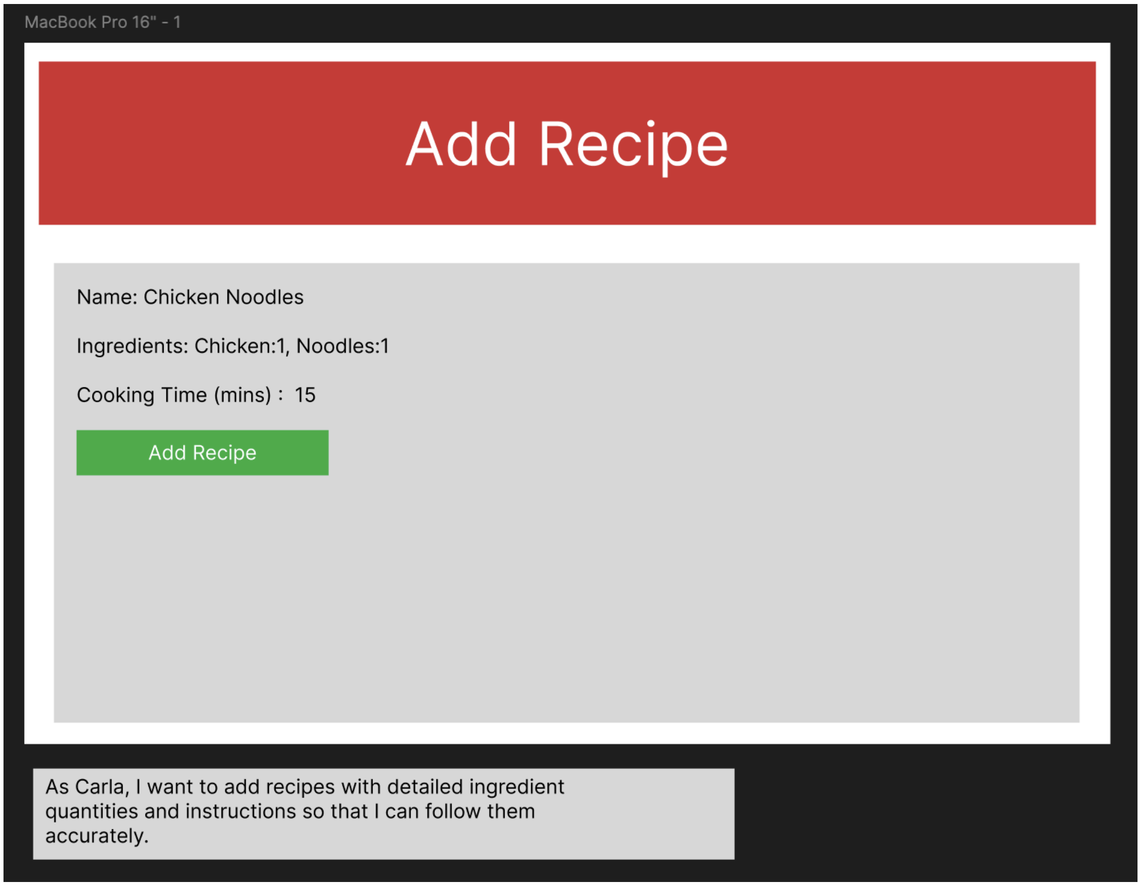
- **Shopping List Screen:**
  Organizes missing or low-stock ingredients into a checklist format, allowing users to mark items as purchased and update the inventory accordingly.



Shopping List

1. Eggs are low, please replenish.
2. Chicken is low please replenish

1. As Emily, I want the app to generate a shopping list for missing ingredients so that I can plan my grocery trips efficiently.
2. As Carla, I want a shopping list that automatically adjusts based on my usage to ensure I have fresh ingredients on hand.



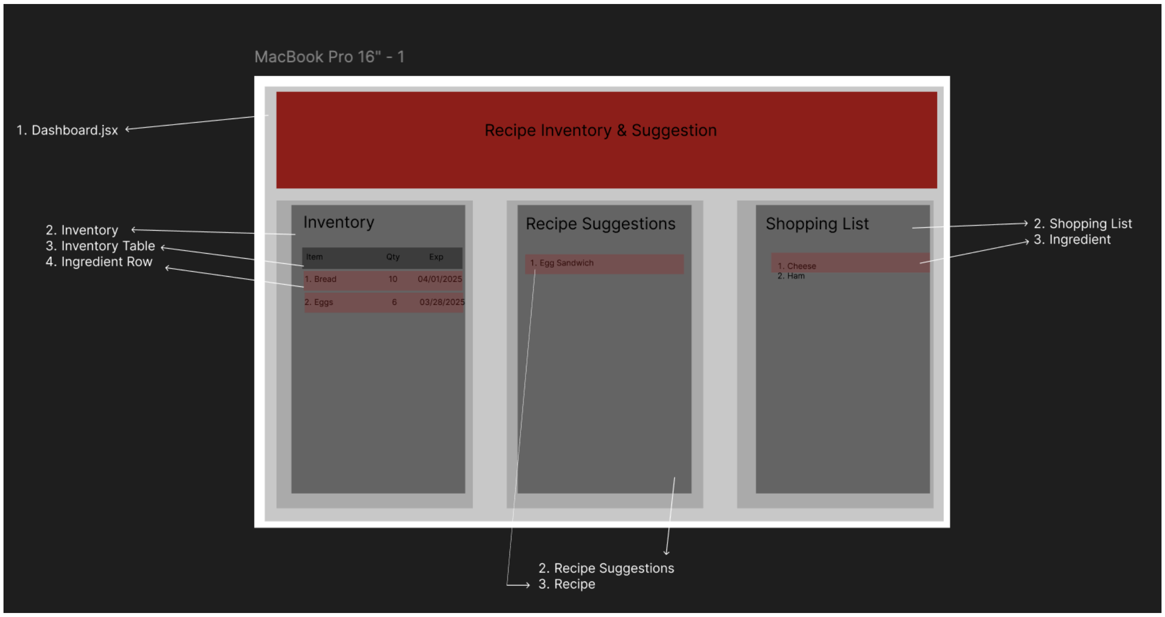
Name: Chicken Noodles

Ingredients: Chicken:1, Noodles:1

Cooking Time (mins) : 15

As Carla, I want to add recipes with detailed ingredient quantities and instructions so that I can follow them accurately.

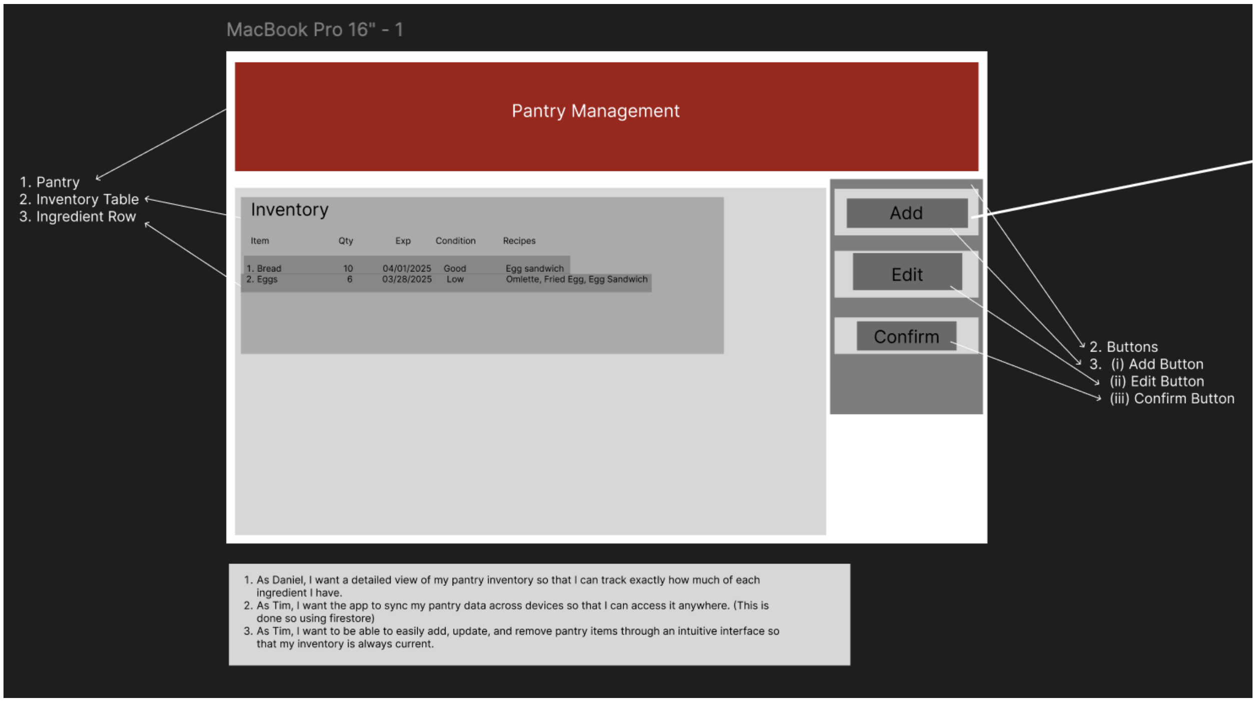
Thinking in React



Item	Qty	Exp
1. Bread	10	04/01/2025
2. Eggs	6	03/28/2025

1. Egg Sandwich

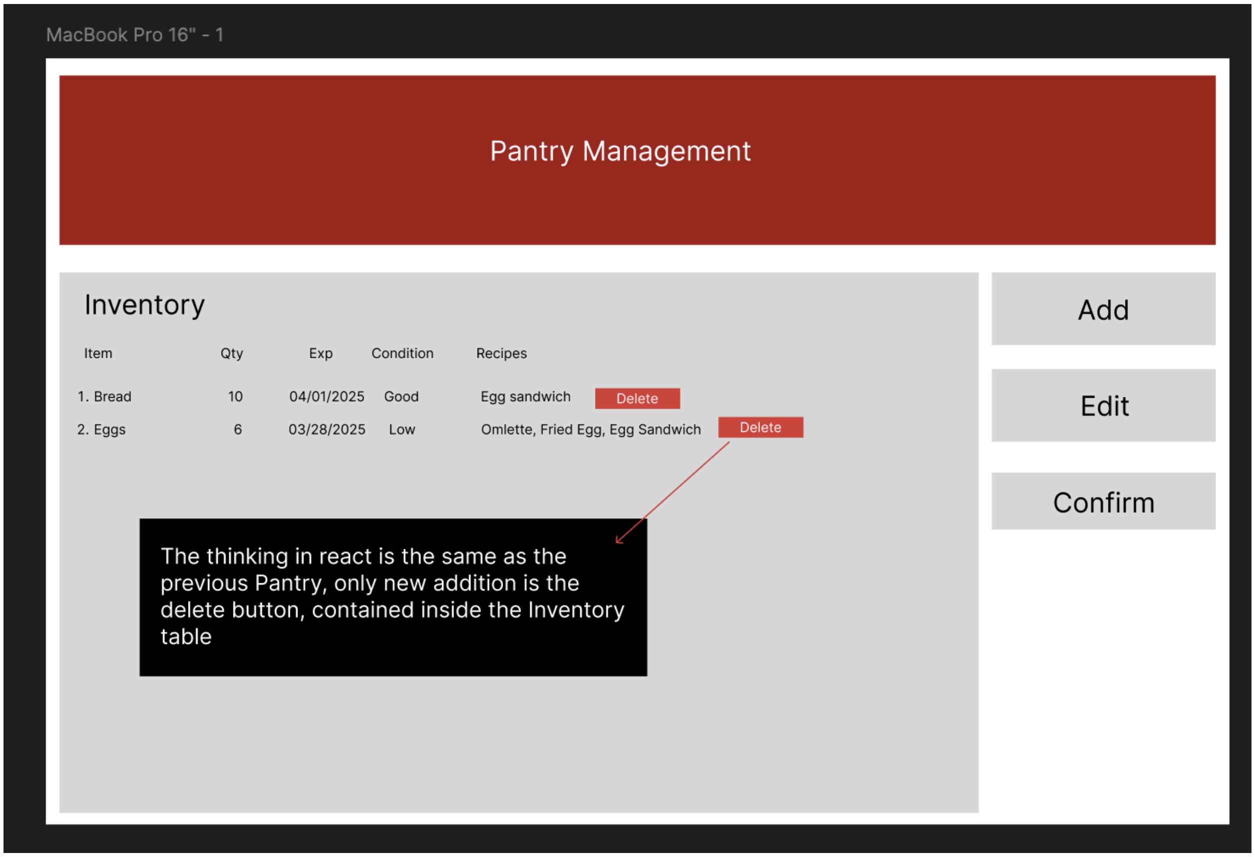
1. Cheese
2. Ham



Inventory

Item	Qty	Exp	Condition	Recipes
1. Bread	10	04/01/2025	Good	Egg sandwich
2. Eggs	6	03/28/2025	Low	Omlette, Fried Egg, Egg Sandwich

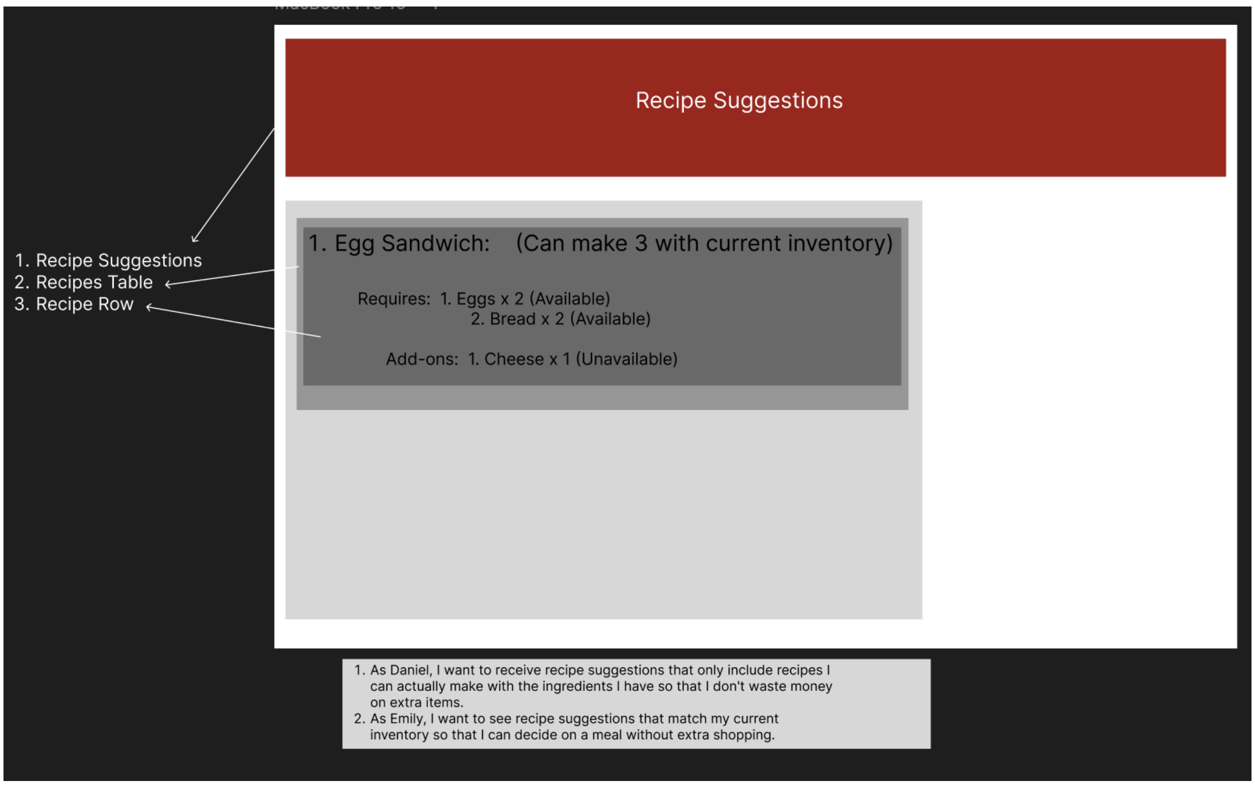
1. As Daniel, I want a detailed view of my pantry inventory so that I can track exactly how much of each ingredient I have.
2. As Tim, I want the app to sync my pantry data across devices so that I can access it anywhere. (This is done so using firestore)
3. As Tim, I want to be able to easily add, update, and remove pantry items through an intuitive interface so that my inventory is always current.



Inventory

Item	Qty	Exp	Condition	Recipes	
1. Bread	10	04/01/2025	Good	Egg sandwich	Delete
2. Eggs	6	03/28/2025	Low	Omlette, Fried Egg, Egg Sandwich	Delete

The thinking in react is the same as the previous Pantry, only new addition is the delete button, contained inside the Inventory table



Recipe Suggestions

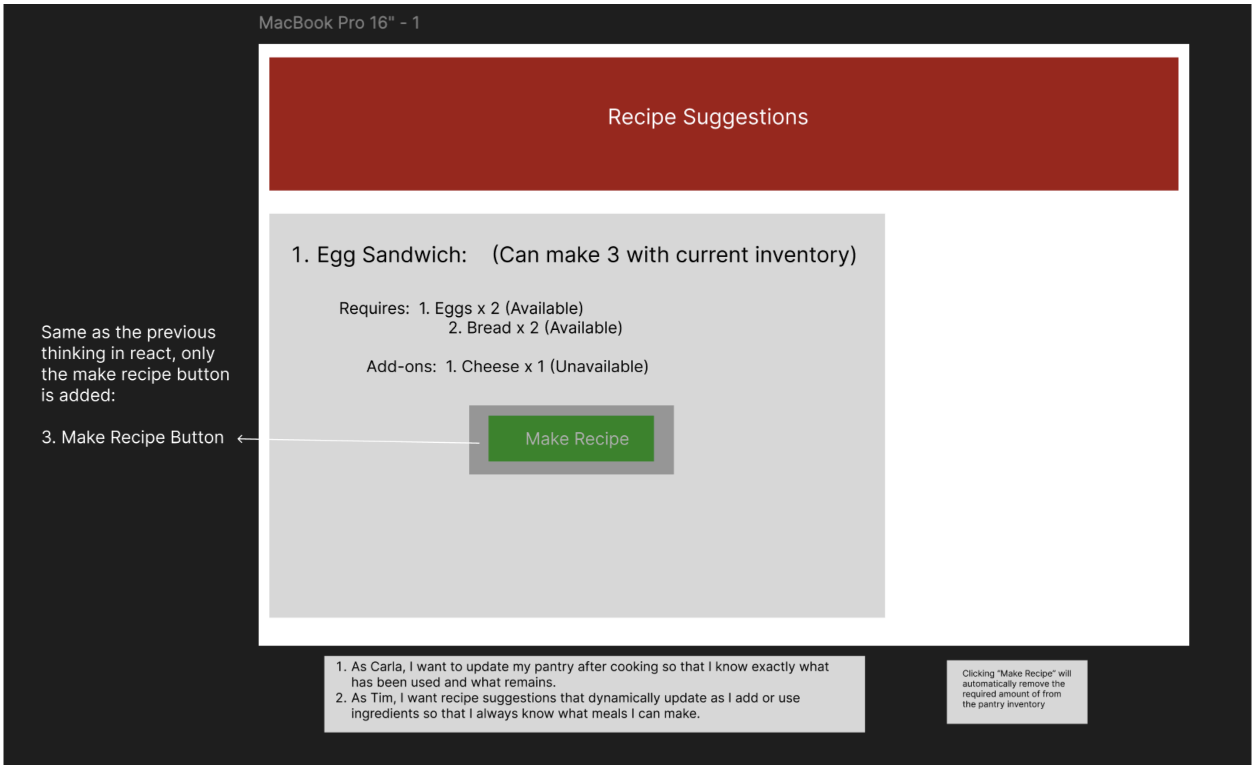
1. Egg Sandwich: (Can make 3 with current inventory)

Requires: 1. Eggs x 2 (Available)
2. Bread x 2 (Available)

Add-ons: 1. Cheese x 1 (Unavailable)

1. Recipe Suggestions
2. Recipes Table
3. Recipe Row

1. As Daniel, I want to receive recipe suggestions that only include recipes I can actually make with the ingredients I have so that I don't waste money on extra items.
2. As Emily, I want to see recipe suggestions that match my current inventory so that I can decide on a meal without extra shopping.



MacBook Pro 16" - 1

Recipe Suggestions

1. Egg Sandwich: (Can make 3 with current inventory)

Requires: 1. Eggs x 2 (Available)
2. Bread x 2 (Available)

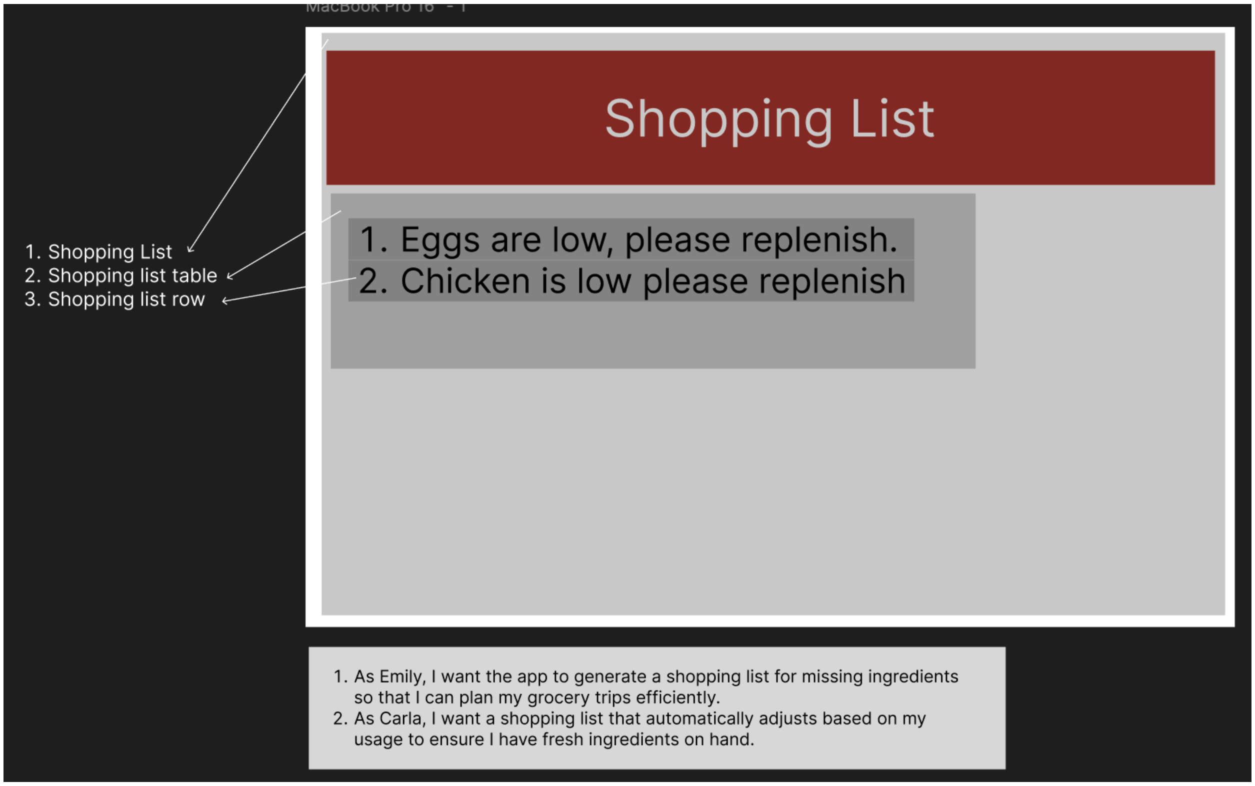
Add-ons: 1. Cheese x 1 (Unavailable)

Same as the previous thinking in react, only the make recipe button is added:

3. Make Recipe Button

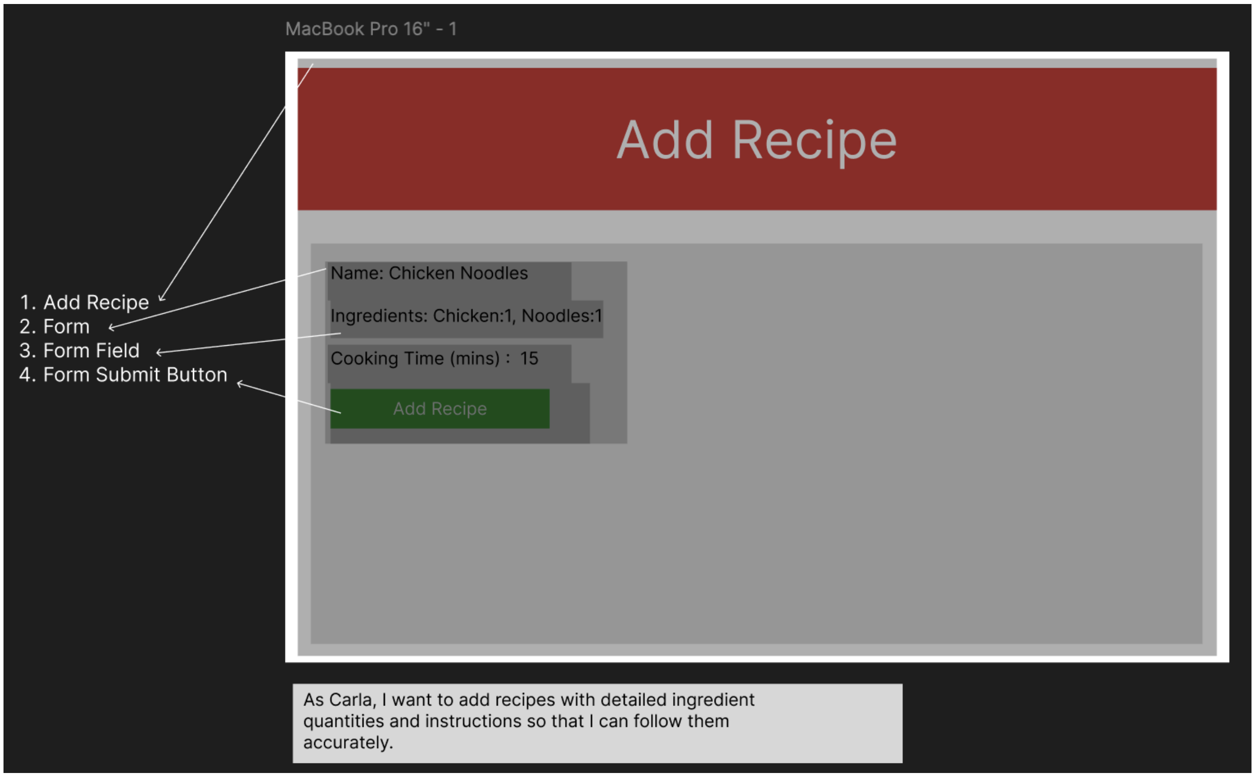
1. As Carla, I want to update my pantry after cooking so that I know exactly what has been used and what remains.
2. As Tim, I want recipe suggestions that dynamically update as I add or use ingredients so that I always know what meals I can make.

Clicking "Make Recipe" will automatically remove the required amount of from the pantry inventory



1. Eggs are low, please replenish.
2. Chicken is low please replenish

1. As Emily, I want the app to generate a shopping list for missing ingredients so that I can plan my grocery trips efficiently.
2. As Carla, I want a shopping list that automatically adjusts based on my usage to ensure I have fresh ingredients on hand.



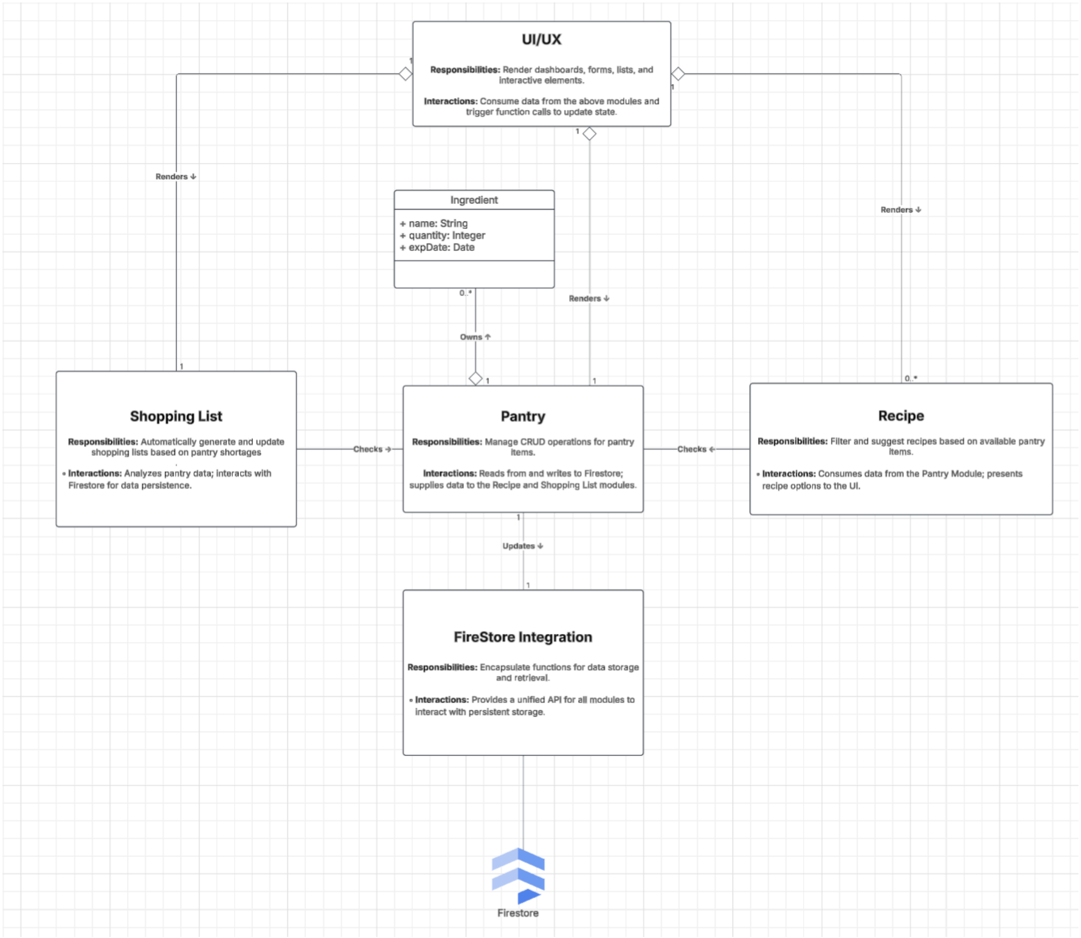
Name: Chicken Noodles

Ingredients: Chicken:1, Noodles:1

Cooking Time (mins) : 15

As Carla, I want to add recipes with detailed ingredient quantities and instructions so that I can follow them accurately.

Modules Diagram (this is the link to the old one - done before coding)



Responsibilities: Render dashboards, forms, lists, and interactive elements.

Interactions: Consume data from the above modules and trigger function calls to update state.

Responsibilities: Automatically generate and update shopping lists based on pantry shortages

- **Interactions:** Analyzes pantry data; interacts with Firestore for data persistence.

Responsibilities: Manage CRUD operations for pantry items.

Interactions: Reads from and writes to Firestore; supplies data to the Recipe and Shopping List modules.

Responsibilities: Filter and suggest recipes based on available pantry items.

- **Interactions:** Consumes data from the Pantry Module; presents recipe options to the UI.

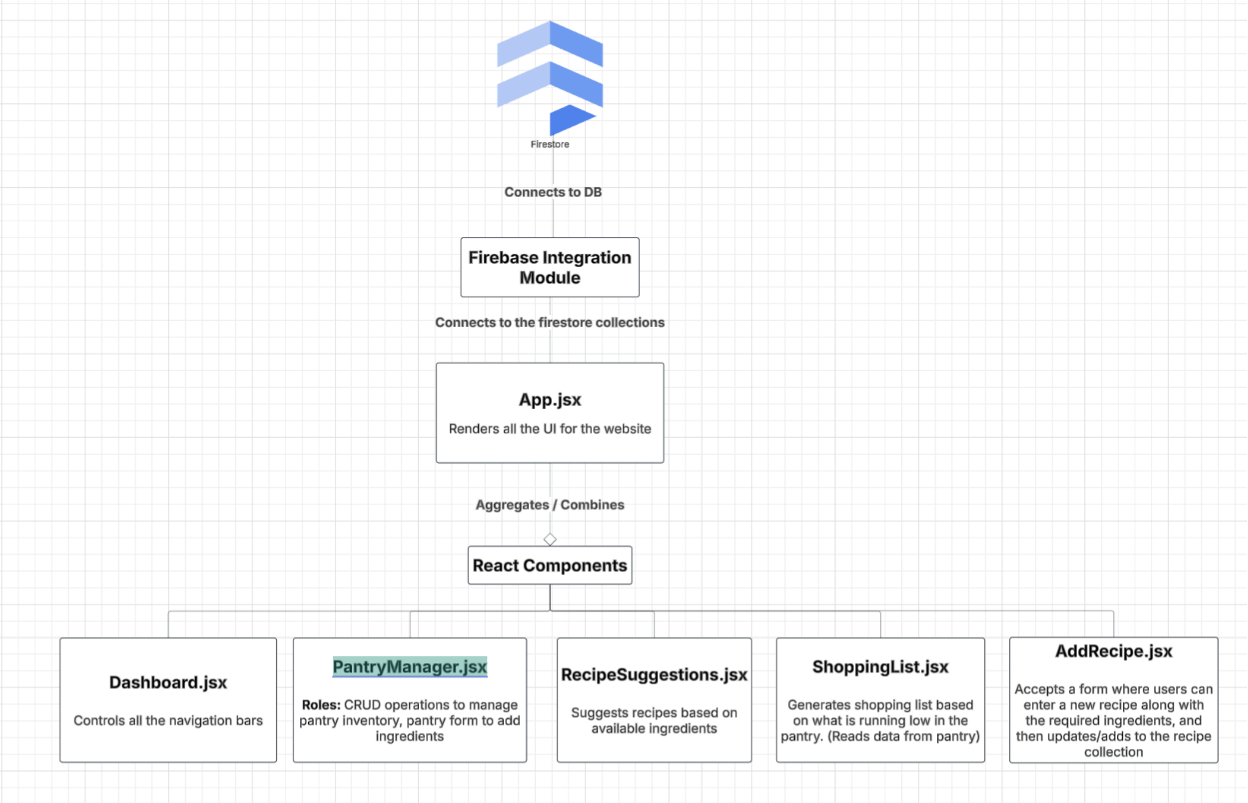
Responsibilities: Encapsulate functions for data storage and retrieval.

- **Interactions:** Provides a unified API for all modules to interact with persistent storage.

- **Pantry Module:**
  - **Responsibilities:** Manage CRUD operations for pantry items.
  - **Interactions:** Reads from and writes to Firestore; supplies data to the Recipe and Shopping List modules.
- **Recipe Module:**
  - **Responsibilities:** Filter and suggest recipes based on available pantry items.
  - **Interactions:** Consumes data from the Pantry Module; presents recipe options to the UI.
- **Shopping List Module:**
  - **Responsibilities:** Automatically generate and update shopping lists based on pantry shortages.
  - **Interactions:** Analyzes pantry data; interacts with Firestore for data persistence.

- **FireStore Integration Module:**
    - **Responsibilities:** Encapsulate functions for data storage and retrieval.
    - **Interactions:** Provides a unified API for all modules to interact with persistent storage.
- **UI/UX Components:**
    - **Responsibilities:** Render dashboards, forms, lists, and interactive elements.
    - **Interactions:** Consume data from the above modules and trigger function calls to update state.

Updated Modules Diagram


Firestore
Connects to DB
Firebase Integration Module
Connects to the firestore collections
App.jsx
Renders all the UI for the website
Aggregates / Combines
React Components
Dashboard.jsx
Controls all the navigation bars
PantryManager.jsx
Roles: CRUD operations to manage pantry inventory, pantry form to add ingredients
RecipeSuggestions.jsx
Suggests recipes based on available ingredients
ShoppingList.jsx
Generates shopping list based on what is running low in the pantry. (Reads data from pantry)
AddRecipe.jsx
Accepts a form where users can enter a new recipe along with the required ingredients, and then updates/adds to the recipe collection

CRUD Operations

Overview:

The application interacts with Firestore to perform the following operations:

- Create: Adding new pantry items and recipes.
- Read: Fetching pantry items, recipes, and shopping list data.
- Update: Modifying existing documents (e.g., updating an ingredient's quantity, including updates made when "making" a recipe).
- Delete: Removing documents from Firestore (e.g., deleting a pantry item).
- Each operation is implemented using Firebase's methods such as addDoc, onSnapshot, updateDoc, and deleteDoc.

Create Operations

1. Adding a New Pantry Item

When a user submits the form in the Pantry Manager, the app first checks if the ingredient already exists (using a Firestore query). If it does not exist, it creates a new document in the pantry collection.

Code Example:

```jsx
await addDoc(collection(db, 'pantry'), {
  name: normalizedName,
  quantity,
});
```

Explanation: This call creates a new document in the pantry collection with the fields name (normalized to lowercase and trimmed) and quantity.

2. Adding a New Recipe

In the AddRecipe component, the user inputs a comma-separated list of ingredients with quantities (e.g., "Bread:2, Eggs:2"). The app parses this string into an array of objects and stores the recipe in the recipes collection.

Code Example:

```
const ingredientsArray = ingredients.split(",").map((item) => {
  const [ingName, qty] = item.split(":").map(str => str.trim());
  return { name: ingName, quantity: Number(qty) };
});
```

```
await addDoc(collection(db, 'recipes'), {
  name,
  ingredients: ingredientsArray,
  cuisine,
  cookingTime: cookingTime ? Number(cookingTime) : null,
});
```

Explanation: This converts the input string into an array of ingredient objects and then uses addDoc to store the new recipe in Firestore.

Read Operations

1. Reading Pantry Items

Pantry items are fetched in real time using an onSnapshot listener in the PantryManager component.

Code Example:

```jsx
useEffect(() => {
  const q = query(collection(db, 'pantry'));
  const unsubscribe = onSnapshot(q, (snapshot) => {
    const items = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      ...docSnap.data(),
    }));
    setIngredients(items);
  });
  return () => unsubscribe();
}, []);
```

Explanation: This code sets up a real-time subscription to the pantry collection. The snapshot is transformed into an array of objects using map, and the component state is updated accordingly.

2. Reading Recipes

Similarly, the RecipeSuggestions component fetches recipes using an onSnapshot listener.

Code Example:

```jsx
useEffect(() => {
  const qRecipes = query(collection(db, 'recipes'));
  const unsubscribeRecipes = onSnapshot(qRecipes, (snapshot) => {
    const recs = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      ...docSnap.data(),
    }));
    setRecipes(recs);
  });
  return () => unsubscribeRecipes();
}, []);
```

Explanation: This listener retrieves and updates the recipe list in real time as changes occur in the Firestore recipes collection.

Update Operations

1. Updating a Pantry Item's Quantity

When an ingredient already exists, the app updates its quantity by summing the current quantity with the new amount provided.

Code Example:

```jsx
const newQuantity = (existingData.quantity || 0) + quantity;
await updateDoc(doc(db, 'pantry', existingDoc.id), {
  quantity: newQuantity,
});
```

Explanation: This updates the specified document in the pantry collection with the new calculated quantity.

2. Updating Pantry Data When "Making a Recipe"

When a user clicks the Make Recipe button, the app updates the pantry by deducting the required quantities of each ingredient used in the recipe.

Code Example:

```jsx
```

```
const makeRecipe = async (recipe) => {
  for (const reqIng of recipe.ingredients) {
    const normalizedReqName = reqIng.name.toLowerCase().trim();
    const pantryItem = pantryItems.find(item => item.name === normalizedReqName);
    if (pantryItem) {
      const newQuantity = pantryItem.quantity - reqIng.quantity;
      await updateDoc(doc(db, 'pantry', pantryItem.id), {
        quantity: newQuantity < 0 ? 0 : newQuantity
      });
      console.log(`Updated ${normalizedReqName} to ${newQuantity < 0 ? 0 : newQuantity}`);
    }
  }
}
```

Explanation: This function loops through each required ingredient in a recipe and finds the corresponding pantry item. It then updates the pantry item's quantity by subtracting the required amount. If the new quantity is negative, it is set to zero. This ensures that "making" a recipe automatically adjusts the inventory.

Delete Operations

1. Deleting a Pantry Item

Users can remove a pantry item by clicking a Delete button, which triggers a deletion from Firestore.

Code Example:

```jsx
await deleteDoc(doc(db, 'pantry', itemId));
```

Explanation: This call removes the document identified by itemId from the pantry collection, ensuring that the UI updates accordingly.

React Hooks

1. useState

- **Purpose:**
  useState is used to declare state variables in functional components. It allows components to hold and update values, such as form inputs or fetched data, and trigger re-renders when state changes.

Usage in the Project

- **Managing Form Inputs:** In components like PantryManager.jsx, AddRecipe.jsx, and others, useState is used to store values such as the ingredient name, quantity, recipe details, etc.
- **Storing Fetched Data:** State variables are used to store arrays of pantry items and recipes retrieved from Firestore.

Example from PantryManager.jsx

```jsx
const [name, setName] = useState('');
const [quantity, setQuantity] = useState(1);
const [ingredients, setIngredients] = useState([]);
```

2. useEffect

- **Purpose:** useEffect lets you perform side effects in functional components. This includes fetching data, setting up subscriptions, or manually updating the DOM. It helps manage the component lifecycle in function components (similar to lifecycle methods in class components).

Usage in the Project

- **Real-Time Data Fetching:** In both PantryManager.jsx and RecipeSuggestions.jsx, useEffect is used with Firebase Firestore's onSnapshot method to subscribe to real-time data changes.
- **Cleanup:** It returns cleanup functions to unsubscribe from listeners when the component unmounts.

Example from PantryManager.jsx

```jsx
useEffect(() => {
  const q = query(collection(db, 'pantry'));
  const unsubscribe = onSnapshot(q, (snapshot) => {
    const items = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      ...docSnap.data(),
    }));
    setIngredients(items);
  });
  return () => unsubscribe();
}, []);
```

- **These hooks allow the app to:**
  - **Maintain Dynamic UI:** By storing and updating state values like form inputs and fetched data.
  - **Handle Side Effects:** By subscribing to real-time data updates and cleaning up after themselves, ensuring optimal performance and resource usage.

Functional Programming Principles in my code

1. Pure Functions

Example in Code: The normalizeIngredient function in our components is a pure function. It takes an input string and returns a normalized (lowercase, trimmed) version without any side effects.

```jsx
function normalizeIngredient(ing) {
  return ing.toLowerCase().trim();
}
```

Why It's Good FP: Every time you call normalizeIngredient(' Eggs '), it will always return "eggs" without modifying any external state or relying on it.

Hypothetical Break: If normalizeIngredient were to modify a global variable (e.g., incrementing a counter every time it’s called) or use a random value in its computation, it would no longer be pure. For example:

```jsx
let counter = 0;
function impureNormalize(ing) {
  counter++; // side effect!
  return ing.toLowerCase().trim() + counter; // result now depends on external state
}
```

This would break the pure function principle.

2. Immutability

Example in Code: When filtering and mapping arrays, we never modify the original array; we create new arrays. In the RecipeSuggestions component, for instance, we use filter to derive a list of suggested recipes:

```jsx
const suggestedRecipes = recipes.filter((recipe) =>
  Array.isArray(recipe.ingredients) &&
  recipe.ingredients.every((ing) =>
    pantryIngredients.includes(normalizeIngredient(ing))
  )
);
```

Why It's Good FP: The original recipes array remains unchanged; we create a new array suggestedRecipes based solely on existing data.

Hypothetical Break: If instead we looped over recipes and modified each recipe object directly (e.g., subtracting quantities directly from the original objects), we would violate immutability. For example:

```jsx
recipes.forEach(recipe => {
  recipe.ingredients.forEach(ing => {
    ing.quantity -= 1; // mutating the object directly
  });
});
```

This would change the original data and lead to bugs and unpredictable state.

3. First-Class Functions

Example in Code: Functions like normalizeIngredient and array methods (map, filter) are passed as values. For instance, in the parsing logic in AddRecipe.jsx, we pass functions to map:

```jsx
```

```
const ingredientsArray = ingredients.split(",").map((item) => {
  const [ingName, qty] = item.split(":").map(str => str.trim());
  return { name: ingName, quantity: Number(qty) };
});
```

Why It's Good FP: This treats functions as first-class citizens by passing them as arguments to other functions, enabling concise, expressive code.

Hypothetical Break: A language or design that disallows passing functions (or treats them as first-class objects) would force you to write less modular, more rigid code (e.g., using switch-case statements or inline loops without helper functions).

4. Higher-Order Functions

Example in Code: Our use of filter, map, and every in the code are classic examples of higher-order functions—they accept functions as arguments to process arrays.

```jsx
const suggestedRecipes = recipes.filter((recipe) =>
  Array.isArray(recipe.ingredients) &&
  recipe.ingredients.every((ing) =>
    pantryIngredients.includes(normalizeIngredient(ing))
  )
);
```

Why It's Good FP: These functions abstract away the iteration logic, allowing us to declaratively state what we want to do (e.g., "filter these recipes") without managing the loop explicitly.

Hypothetical Break: If you had to write a manual loop (imperatively) with index counters and conditional checks each time, you'd lose the abstraction and conciseness provided by higher-order functions. This would make the code more error-prone and harder to read.

5. Declarative over Imperative

Example in Code: Our React components are written using JSX, which is declarative. We describe what the UI should look like based on the state, not how to update the DOM step-by-step.

```jsx
return (
  <div>
```

```
    <h2>Recipe Suggestions</h2>
    {suggestedRecipes.length > 0 ? (
      <ul>
        {suggestedRecipes.map((recipe) => (
          <li key={recipe.id}>{recipe.name}</li>
        ))}
      </ul>
    ) : (
      <p>No recipes available with current pantry items.</p>
    )}
  </div>
);
```

Why It's Good FP: Declarative code is easier to understand and maintain because it describes the desired state of the UI rather than the process to get there.

Hypothetical Break: If you were to manipulate the DOM directly using imperative methods like document.getElementById() and manually updating innerHTML, your code would become more complex and harder to maintain, which is contrary to the declarative nature of React.

Array Functional Programming Methods Examples

1. Mapping Firestore Documents:

In our data fetching code, we use map to transform Firestore documents into usable JavaScript objects:

```jsx
const items = snapshot.docs.map((docSnap) => ({
  id: docSnap.id,
  ...docSnap.data(),
}));
```

2. Filtering Recipes:

We use filter (combined with every) to select only those recipes that have all required ingredients available:

```jsx
const suggestedRecipes = recipes.filter((recipe) =>
  Array.isArray(recipe.ingredients) &&
  recipe.ingredients.every((ing) =>
    pantryIngredients.includes(normalizeIngredient(ing))
  )
);
```

3. Accumulating items (using reduce)

We use reduce to calculate the total quantity of pantry items in the PantryManager component.

```jsx
const totalQuantity = ingredients.reduce(
  (total, item) => total + item.quantity,
  0
);
```

Design Patterns in Our Code

1. Singleton Pattern (Firebase Initialization):

The Firebase configuration file ensures that only one instance of the Firebase app and Firestore database is created:

```jsx
import { initializeApp } from 'firebase/app';
import { getFirestore } from 'firebase/firestore';

const firebaseConfig = { /* configuration */ };
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);

export { db };
```

~~~
```
~~~

Why It's Good: It guarantees that throughout the entire application, the same Firebase instance is used.

Hypothetical Break: If you reinitialize Firebase in multiple modules (e.g., calling initializeApp repeatedly), you could end up with conflicting instances or wasted resources.

2. Observer Pattern (Real-Time Updates with onSnapshot):

Components subscribe to changes in Firestore via onSnapshot, which is a classic Observer pattern implementation:

~~~
```jsx
useEffect(() => {
  const q = query(collection(db, 'pantry'));
  const unsubscribe = onSnapshot(q, (snapshot) => {
    const items = snapshot.docs.map(docSnap => ({
      id: docSnap.id,
      ...docSnap.data(),
    }));
    setIngredients(items);
  });
  return () => unsubscribe();
}, []);
```
~~~

Why It's Good: This pattern keeps the UI in sync with the database in real time without manual polling.

Hypothetical Break: If you replaced this with a manual polling mechanism (e.g., setInterval to fetch data), you’d lose the efficiency and responsiveness of the Observer pattern.

3. Module Pattern (Component-Based Architecture):

Our project is organized into separate modules (files) where each component encapsulates its own functionality and exports a single function:

~~~
```jsx
// PantryManager.jsx
function PantryManager() {
  const [name, setName] = useState('')
  const [quantity, setQuantity] = useState(1)
  const [ingredients, setIngredients] = useState([])
~~~

```
// Real-time listener to fetch pantry items
useEffect(() => {
  const q = query(collection(db, 'pantry'))
  const unsubscribe = onSnapshot(q, (snapshot) => {
    const items = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      ...docSnap.data(),
    }))
    setIngredients(items)
  })

  return () => unsubscribe()
}, [])

// Add or update an ingredient in Firestore
const addIngredient = async (e) => {
  e.preventDefault()
  if (!name.trim()) return

  // Normalize the name to lowercase and trim whitespace
  const normalizedName = name.trim().toLowerCase()

  try {
    // Check if the ingredient already exists in Firestore
    const qCheck = query(
      collection(db, 'pantry'),
      where('name', '==', name)
    )
    const querySnapshot = await getDocs(qCheck)

    if (!querySnapshot.empty) {
      // If found, update its quantity
      const existingDoc = querySnapshot.docs[0]
      const existingData = existingDoc.data()
      const newQuantity = (existingData.quantity || 0) + quantity

      await updateDoc(doc(db, 'pantry', existingDoc.id), {
        quantity: newQuantity,
      })
      console.log(`Updated "${normalizedName}" to quantity ${newQuantity}`)
    } else {
      // If not found, add a new document
      await addDoc(collection(db, 'pantry'), {
        name: name,
        quantity,
      })
      console.log(`Added new ingredient: "${normalizedName}"`)
    }

    // Reset form fields
    setName('')
    setQuantity(1)
  } catch (error) {
    console.error('Error adding/updating ingredient:', error)
  }
}

// Delete an ingredient from Firestore
const deleteIngredient = async (id) => {
  try {
    await deleteDoc(doc(db, 'pantry', id))
    console.log(`Deleted ingredient with id: ${id}`)
  } catch (error) {
    console.error('Error deleting ingredient:', error)
  }
}

return (
  <div>
    <h2>Pantry Manager</h2>
    <form onSubmit={addIngredient}>
      <input
        type="text"
        placeholder="Ingredient Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
        required
      />
      <input
        type="number"
        min="1"
        placeholder="Quantity"
        value={quantity}
        onChange={(e) => setQuantity(Number(e.target.value))}
        required
      />
      <button type="submit">Add Ingredient</button>
    </form>
```

```
      <h3>Pantry Items</h3>
      <ul>
        {ingredients.map((item) => (
          <li key={item.id}>
            {item.name} - {item.quantity}{' '}
            <button onClick={() => deleteIngredient(item.id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  )
}

export default PantryManager
```

Why It's Good: This design promotes separation of concerns, reusability, and maintainability. Each module has a well-defined interface.

Hypothetical Break: If all the code were placed in a single file with no separation (monolithic code), it would become difficult to manage, test, and reuse components, breaking the pattern.

AI Usage

1. Formatting User Stories and Personas

Model: GPT-4

Prompt:
"Please format the user stories and personas for the Recipe Ingredient Inventory & Suggestion App, based on my existing project document. Include user dimensions, detailed user stories, and ensure the personas reflect both novice and expert aspects."
I uploaded my document with my updated user stories and personas

(very badly formatted T_T).

Response:

Use Cases
Use Case 1: Pantry Inventory Management
User Story 1:
As a user, I want to add ingredients to my pantry so that I can track what I have available.

User Story 2:
As a user, I want to update the quantity or details of an ingredient so that my inventory stays accurate.

User Story 3:
As a user, I want to delete an ingredient from my pantry when it's no longer available so that my list remains current.

Use Case 2: Recipe Suggestions
User Story 1:
As a user, I want the app to suggest recipes based on the current ingredients in my pantry so that I can quickly decide what to cook.

User Story 2:
As a user, I want to see the required ingredients and quantities for each recipe so that I know if I have enough supplies.

User Story 3:
As a user, I want to click a “Make Recipe” button that deducts the used quantities from my pantry so that the app helps me manage inventory automatically.

Use Case 3: Shopping List Generation
User Story 1:
As a user, I want the app to automatically generate a shopping list for ingredients that are low or missing so that I can easily plan my grocery shopping.

User Story 2:
As a user, I want to be able to mark items on the shopping list as purchased to keep my inventory up-to-date.

User Story 3:
As a user, I want the shopping list to update in real time as my pantry inventory changes so that I always know what I need.

Use Case 4: Real-Time Data Synchronization
User Story 1:
As a user, I want my pantry data, recipe suggestions, and shopping lists to be synced across all my devices so that I can access them anytime.

User Story 2:
As a user, I want any changes I make in the app to update immediately so that my inventory is always accurate.

User Story 3:
As a user, I want the app to function seamlessly in real time so that I never have to manually refresh data.

User Stories

Persona 1: Emily – The Busy Professional (Novice Cook, Novice Techie)
Profile:
Emily works long hours, rarely has time to plan meals, and often finds herself missing a key ingredient. She needs an app that is straightforward and quick to update.

User Stories:

As Emily, I want to quickly add ingredients to my pantry so that I know what I have at any moment.

As Emily, I want to see recipe suggestions that match my current inventory so that I can decide on a meal without extra shopping.

As Emily, I want the app to generate a shopping list for missing ingredients so that I can plan my grocery trips efficiently.

Persona 2: Daniel – The Budget-Conscious Shopper (Novice Cook, Expert Tech)
Profile:
Daniel is focused on reducing waste and saving money. Although he isn't a skilled cook, he uses technology to keep track of his ingredients and avoid unnecessary purchases.

User Stories:

As Daniel, I want a detailed view of my pantry inventory so that I can track exactly how much of each ingredient I have.

As Daniel, I want to update or delete items easily so that my pantry data remains accurate over time.

As Daniel, I want to receive recipe suggestions that only include recipes I can actually make with the ingredients I have so that I don't waste money on extra items.

Persona 3: Carla – The Aspiring Chef (Expert Cook, Moderate Tech)
Profile:
Carla enjoys cooking and experimenting with recipes but values organization in the kitchen. She wants to efficiently manage her pantry to support her culinary creativity.

User Stories:

As Carla, I want to add recipes with detailed ingredient quantities and instructions so that I can follow them accurately.

As Carla, I want to update my pantry after cooking so that I know exactly what has been used and what remains.

As Carla, I want a shopping list that automatically adjusts based on my usage to ensure I have fresh ingredients on hand.

Persona 4: Tim – The Tech-Savvy Foodie (Novice Cook, Expert Tech)
Profile:

Tim loves exploring new technologies and uses apps to optimize his daily life, including meal planning. Although he isn’t an expert cook, he appreciates the efficiency and smart suggestions provided by modern apps.

User Stories:

As Tim, I want the app to sync my pantry data across devices so that I can access it anywhere.

As Tim, I want to be able to easily add, update, and remove pantry items through an intuitive interface so that my inventory is always current.

As Tim, I want recipe suggestions that dynamically update as I add or use ingredients so that I always know what meals I can make.

2. Website Deployment Assistance

Model: GPT-4

Prompt:
"How do I deploy my Recipe Ingredient Inventory & Suggestion App using Firebase (or similar service)? Provide step-by-step instructions."

Response:

To deploy your Recipe Ingredient Inventory & Suggestion App using Firebase, follow these steps:

1. Install Firebase CLI:

Run: `npm install -g firebase-tools`

2. Login to Firebase:
Run: `firebase login`

3. Initialize Firebase in Your Project:

In your project directory, run: `firebase init`

Select "Hosting" as the feature.

Choose your Firebase project from the list.

Set the build directory as "dist" (the output folder for your production build with Vite).

Configure as a single-page app (typically answer "Yes").

4. Build Your React App:

Run: npm run build

This will create a production build of your app in the "dist" folder.

5. Deploy to Firebase Hosting:

Run: firebase deploy -- only hosting

This command uploads your production build to Firebase Hosting.

3. Project Folder Setup Guidance

Model: GPT-4

Prompt:
"Show me how to set up my project folder using Vite with React and integrate Firebase for the Recipe Ingredient Inventory & Suggestion App."
Response: Your project structure should look like this:

```
recipe-inventory-app/
├── index.html          // in the public folder (or at the project root for Vite)
├── package.json
├── vite.config.js
├── src/
│   ├── main.jsx
│   ├── App.jsx
│   ├── index.css
│   ├── firebase-config.js
│   └── components/
│       ├── Dashboard.jsx
│       ├── PantryManager.jsx
│       ├── RecipeSuggestions.jsx
│       └── ShoppingList.jsx
```

4. AI Documentation

Model: GPT-4

Prompt:
“Could you generate an AI Usage documentation for this project, show me every prompt and response in this chat - format it as (Model, Prompt and Response)”

Response:

- **The above AI Usage declarations**