NutriFind Project Specification

General Description of the Project

NutriFind is a web-based application that enables users to discover foods that align with their specific dietary needs and preferences. The app provides a comprehensive nutrition database, allowing users to filter foods based on criteria like low-carb, high-protein, gluten-free, or vegan. NutriFind simplifies the process of finding suitable foods, making it easier for users to plan meals that adhere to their health goals and dietary restrictions.

The application will be built using Python and Flask to handle backend processing and user input. Nutritional data and recipes will be sourced using the Edamam API, which provides comprehensive information about food items, their nutrients (calories, protein, carbs, fats, etc.), and diet compatibility (e.g., vegan, gluten-free). The API returns data in JSON format, making it easy to process and display in the application. The app's frontend will be developed using HTML and CSS for a clean and user-friendly interface. The backend will handle user requests and render the appropriate results. This combination of technologies allows NutriFind to be both scalable and efficient in processing user requests.

The ideal GUI will feature an intuitive layout where users select their dietary preferences using interactive dropdowns, checkboxes, and sliders. While the primary version is web-based, removed CLI reference. This feature is being dropped due to time constraints. The project will focus solely on the web-based interface to maintain feasibility. In the future, the app could be extended to include a REST API, enabling third-party applications to access the database and filtering functionalities remotely.

Task Vignettes

1. **Setting Dietary Preferences:**
   Sarah is looking to maintain a gluten-free and high-protein diet. She opens NutriFind and sees a form with a variety of dietary preferences such as gluten-free, low-carb, dairy-free, etc. Sarah selects "Gluten-Free" and "High-Protein" from the options and clicks the "Filter Foods" button.

   Technical Details:
   - User input is captured through a series of checkboxes and dropdowns.
   - The selected preferences are sent to the Flask backend via an HTTP request.
   - The backend uses the filtering logic to process the input against the nutrition database.

2. **Viewing Filtered Results:**
   After Sarah submits her dietary preferences, NutriFind processes the data and presents her with a list of foods that match her criteria. Each item on the list shows detailed nutritional information, including calories, protein content, vitamins, and minerals. Sarah clicks on "Quinoa" to see more details.

   Technical Details:
   - The backend retrieves data from the nutrition database using the user's input criteria.
   - Filtered results are returned as a response to the frontend.
   - The filtered results are displayed on the results page after the backend processes the user's input and sends the data to the frontend.

3. **Refining Search Criteria:**
   Sarah decides she also wants to see only foods with low sugar content. She updates her search by selecting "Low-Sugar" from the options and clicks "Update Results." The user submits a new search request to update the list of results based on additional dietary restrictions, which are then rendered by Flask on the results page.

   Technical Details:
   - The filtering process is triggered again, incorporating the new dietary restriction.
   - The updated results are rendered by Flask and displayed on the results page after the user submits the updated criteria.

4. **Saving Dietary Preferences:**
   Sarah wants to save her dietary preferences for future use. She clicks the "Save Preferences" button, and the app prompts her to create an account or log in. Once she logs in, her preferences are saved, allowing for easy retrieval next time she uses the app.

   Technical Details:
   - User preferences are stored in a database linked to their account.
   - Flask manages the user authentication and data storage.
   - The frontend offers an option to load saved preferences upon login.

Technical Flow

1. User Input: Users select dietary preferences using the GUI. These inputs (lists of selected criteria) are sent to the Flask server via HTTP requests.

2. Data Retrieval & Processing: Flask processes the request and queries the comprehensive nutrition database (likely stored in a SQL or NoSQL database) to retrieve foods that match the selected criteria. The data is filtered using a Python algorithm that compares each food item against the user's preferences.

3. Data Presentation: The filtered data is converted to JSON format and sent back to the frontend. The GUI displays the results in a clean, tabular format using HTML/CSS, rendered by Flask after processing the user's preferences.

Data Types:

- Input: User-selected preferences (dictionaries/lists).
- Database: A structured table of foods (SQL database), each entry containing fields like food name, calories, protein, carbs, fats, vitamins, minerals, etc.
- Output: Filtered list of food items in JSON format.

Data Flow Diagram (Conceptual)
User Interface → User Preferences → Flask Backend → Filtering Logic → Nutrition Database

Data types:

- User preferences: List of dietary restrictions (e.g., ["gluten-free", "high-protein"])
- Database entries: Dictionary with keys for nutrient values (e.g., {"name": "quinoa", "protein": 14g, "gluten-free": True})
- Output: JSON array of matched foods

Additional Features and Considerations:

- Remote Control/API: A REST API for third-party integration is marked as a nice-to-have feature. If time permits after core functionalities are completed, this will be considered.
- Localization: The application could adapt to different units (e.g., metric vs. imperial) based on user location or preferences.

Biggest Implementation Challenges:
I can work on designing and developing the user interface, including creating the HTML/CSS layout and the filtering form. With guidance, I can also help implement the filtering logic using Python and Flask, ensuring that the application accurately processes user inputs and displays relevant results. Deploying the

application on a web platform like GitHub Pages will finalize the project, making it available to users.
This spec provides a clear blueprint for NutriFind, a user-friendly app that offers personalized food recommendations based on specific dietary needs and preferences.

Github: https://github.com/uma01234/nutrifind

Summary of Changes:

Text in red is the edits that I have made.

1. Data Source (Revised from USDA to Edamam API):

- **Why it Won't Work**: The original spec mentioned using the USDA FoodData Central as the data source for nutritional information. However, integrating multiple datasets or a more complex data source like USDA was not feasible given the time constraints and the implementation of the Edamam API.
- **Salvaging It**: Instead of juggling multiple data sources, I salvaged the functionality by fully committing to the Edamam API, which offers both recipes and nutritional data in an easy-to-use JSON format. This keeps the project feasible without sacrificing core functionality.
- **Conclusion**: The USDA FoodData Central was dropped in favor of the Edamam API, which simplifies implementation.

2. Dynamic JavaScript Interactions (Dropped):

- **Why it Won't Work**: The original specification included dynamic updates to the search results (e.g., updating results without reloading the page using JavaScript). Given the complexity of managing both frontend and backend updates dynamically within the remaining time, this part of the spec was deemed too complex to complete efficiently.
- **Salvaging It**: Instead of using JavaScript for dynamic content updates, the project will rely on Flask to handle full page rendering. Each user search will trigger a page reload where Flask will render the results. This approach is simpler and ensures the core functionality remains intact without over-complicating the implementation.
- **Conclusion**: JavaScript-based dynamic updates were dropped. Flask will handle backend rendering for simplicity.

3. Command Line Interface (CLI) (Dropped):

- **Why it Won't Work**: Initially, a Command Line Interface (CLI) version was mentioned as a secondary way for users to interact with the app. Given the focus on building a web interface and the remaining time, developing a CLI would detract from finalizing the web version.
- **Salvaging It**: The CLI has been dropped entirely to focus efforts on completing the web interface. Salvaging it in the short term would require duplicating functionality already present in the web interface, which is not necessary for this phase of the project.
- **Conclusion**: The CLI feature was dropped to maintain focus on the core web-based functionality.

4. REST API for Third-Party Integration (Deferred as Optional):

- **Why it Won't Work Right Now**: Building a REST API that third-party applications can use to query the database is a more advanced feature, requiring significant additional backend work (handling API requests, securing data, etc.). Given the current timeline, this feature is beyond the scope of what's feasible to implement.
- **Salvaging It**: Instead of dropping this feature entirely, it has been moved to the list of optional features. If the core functionality is completed on time, this feature can be considered as a stretch goal.
- **Conclusion**: The REST API was marked as optional, allowing it to be considered later if time permits.

5. Complex UI Elements (Simplified):

- **Why it Won't Work**: The original spec described a more complex user interface with checkboxes, sliders, and interactive elements for selecting dietary preferences. Given the timeline and the need for a functional MVP (Minimum Viable Product), developing an elaborate UI would take too much time.
- **Salvaging It**: The UI has been simplified to include a basic search box and a dropdown menu for selecting dietary preferences. This keeps the user interface clean and functional while ensuring the project can be completed on time.
- **Conclusion**: Complex UI elements were simplified in favor of a streamlined interface to ensure usability without excessive development effort.