Dietary Recommendation System Documentation

# Overview

The Dietary Recommendation System is an AI-powered web application that generates personalized diet plans based on a user’s health conditions and food allergies. It leverages medical literature from PubMed and generates responses using Google's Gemini AI model. This application helps users receive safe, relevant, and evidence-backed dietary suggestions tailored to their personal health profile.  
  
Core Functionalities:  
- PubMed Article Scraping: Retrieves relevant research abstracts based on user conditions.  
- AI-Powered Recommendation: Uses Gemini AI to interpret medical literature and user input to generate a structured diet plan.  
- Allergy Filtering: Ensures that diet plans do not include foods the user is allergic to.  
- Web-based Interface: Clean and interactive frontend for user input and display of results.

# AI Model

Gemini 2.0 Flash by Google:  
- Provider: Google  
- Model Type: Large Language Model (LLM)  
- Purpose: Used to analyze health data and abstracts, then generate tailored dietary recommendations.  
- Strengths: Fast, cost-efficient, optimized for real-time prompt completion.  
- Accuracy: High performance in text-based health reasoning, depending on input quality and prompt structure.

# Project Structure

| File | Purpose |  
|------|---------|  
| .env | Stores API keys and environment variables securely |  
| config.py | Loads the Gemini API key using dotenv |  
| function.py | Contains logic for scraping PubMed and generating diet recommendations using Gemini |  
| route.py | Defines API routes using FastAPI |  
| main.py | Starts the FastAPI app and configures CORS |  
| App.js | React frontend for user interaction |  
| App.css | CSS styles for UI presentation |  
| requirements.txt | Lists all backend Python dependencies |

# Technical Workflow

1. User Input (Frontend):  
 - User submits their health conditions and known allergies via a React form.  
  
2. API Call (Backend):  
 - The frontend sends this data to the FastAPI backend via a POST request.  
  
3. Web Scraping (Backend):  
 - The backend uses requests and BeautifulSoup to scrape abstracts from PubMed related to the health issues.  
  
4. AI Generation (Backend):  
 - The abstracts and user data are used to build a prompt for Gemini AI, which returns a structured diet plan.  
  
5. Response Rendering (Frontend):  
 - The recommendation is returned as formatted HTML and displayed on the frontend.

# API Endpoints

POST /recommend  
- Input: JSON body with:  
 {  
 "user\_input": "Diabetes, Hypertension",  
 "allergies": ["Dairy", "Nuts"]  
 }  
- Output: A structured diet recommendation string (HTML friendly) based on input.

# Frontend Features

- Built using React.js  
- Real-time feedback with loading spinner  
- Friendly UI with clean typography and spacing  
- Shows result with bullet points and clear formatting  
- Displays helpful messages for errors (e.g., connection issues)

# Testing Tools

Postman:  
- Used for backend endpoint testing before frontend integration  
- Send POST requests with JSON data  
- Inspect response formatting and content

# Performance

| Metric | Value |  
|--------|-------|  
| RAM Usage | Lightweight; depends on Gemini response load |  
| Backend Response Time | ~1-2 seconds average per request |  
| PubMed Scraping | Top 3 articles retrieved per query |  
| Gemini Model Used | models/gemini-2.0-flash |  
  
Note: The recommendation quality depends on the availability of relevant research articles for the provided condition(s).

# Dependencies

Backend (requirements.txt):  
- fastapi – API framework  
- uvicorn – ASGI server  
- python-dotenv – Load .env configs  
- google-generativeai – Gemini API SDK  
- beautifulsoup4 – Web scraping  
- requests – HTTP requests  
  
Frontend (React):  
- React  
- Axios – for making HTTP calls to backend

# Security & Error Handling

- Environment Variables: Gemini API keys are secured using .env files.  
- CORS Middleware: Ensures secure communication between frontend and backend.  
- Error Messaging: Displays fallback messages on network or API failure.  
- Robust Input Parsing: Prevents processing of empty or malformed input.

# Modular Code Structure

- All logic is cleanly separated:  
 - Configuration  
 - Core functions  
 - API routes  
 - Frontend UI  
- This improves maintainability, scalability, and readability of the project.

# Conclusion

The Dietary Recommendation System showcases how real-time AI and medical research can be combined to build intelligent and health-conscious applications. It ensures safety through allergy filtering and leverages large language models for contextual understanding.  
  
This project reflects a strong integration of AI, web development, and health domain understanding — offering a practical solution that’s both useful and technically robust.