**Meal & Calory Planner - Project Documentation**

**1. Introduction**

The *Meal & Calory Planner* is a Django-based web application designed to help users plan meals and fitness routines based on their dietary needs and health goals. The application integrates external nutrition APIs and AI-powered recommendations to offer personalized diet and workout plans.

**2. System Overview**

This project combines the following technologies:

* **Django**: For web framework and backend logic
* **API Ninjas**: For food nutritional data
* **Google Gemini Pro AI**: To generate custom diet and workout suggestions based on user input

**3. Installation and Setup**

**3.1 Prerequisites**

* Python 3.8+
* Pip
* Virtualenv (optional but recommended)

**3.2 Installation Steps**

1. Clone the repository:

git clone <project\_url>

cd Meal-and-calory-planner

1. Create and activate a virtual environment:

python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

1. Install dependencies:

pip install -r requirements.txt

1. Run migrations:

python manage.py migrate

1. Run the development server:

python manage.py runserver

1. Open your browser and go to:

http://127.0.0.1:8000/

**4. Features**

**4.1 Food Nutrition Lookup**

* Users can search for any food item.
* The app fetches nutrition details using API Ninjas.
* Displays macro/micro nutrients and total caloric values.

**4.2 Personalized Diet and Fitness Recommendations**

* Form input includes:
  + Dietary Preferences
  + Fitness Goal
  + Lifestyle Factor
  + Health Conditions
  + Dietary Restrictions
  + Free-text Query
* Uses **Google Gemini AI** to return:
  + 5 Diet Types
  + 5 Workout Routines
  + Meal Suggestions (Breakfast, Dinner)
  + Extra Health Tips

**5. External API Integration**

**5.1 API Ninjas (Nutrition)**

* Endpoint: https://api.api-ninjas.com/v1/nutrition?query=<food>
* Headers: X-Api-Key
* Returns: JSON object with nutrient data per food item

**5.2 Google Gemini Pro AI**

* Library: google.generativeai
* Model: gemini-1.5-pro
* Usage: Prompt-based content generation using user profile data

**6. Application Flow**

1. User accesses homepage
2. Submits a food query
3. App fetches nutrition info from API Ninjas
4. User goes to recommendation form
5. Submits personal details and fitness goal
6. App uses Gemini API to generate diet/workout plan
7. Plan is parsed and displayed in categories

**7. Project Structure (Simplified)**

Meal-and-calory-planner/

|│-- manage.py

|│-- requirements.txt

|├-- foodie/ # Django settings & URLs

|└-- counter/ # Core app logic

|│-- views.py # API interaction, logic

|│-- urls.py # URL routing

|│-- templates/ # HTML templates

|│-- models.py # (Currently empty)

|│-- langchain\_backend.py # (Custom logic)

**8. Future Improvements**

* Add persistent user accounts and authentication
* Save history of diet plans and food searches
* Enhance error handling and edge case input validation
* Visualize nutritional data with charts/graphs

**9. License**

This project is for educational purposes only and may require keys from Google and API Ninjas to function fully in production.

Great — for **Phase 2: Project Selection and Requirements Analysis** of your Meal & Calory Planner project, here’s a full structured response to match academic/industrial software engineering documentation standards.

**✅ Phase 2: Project Selection and Requirements Analysis (Week 1 of 4)**

**📌 1. Project Selection Summary**

**Project Title**: *Meal & Calory Planner*

**Brief Description**:  
The Meal & Calory Planner is a web-based application that leverages nutritional APIs and AI models to provide users with meal suggestions, calorie breakdowns, and personalized fitness recommendations based on their input data (e.g., dietary preferences, fitness goals).

**📍 2. System Requirements**

**🔹 Functional Requirements (FR)**

1. **FR1**: The system shall allow users to input a food item and fetch its nutritional breakdown.
2. **FR2**: The system shall collect user data (preferences, goals, health conditions) via a form.
3. **FR3**: The system shall send prompts to Google Gemini AI and retrieve a customized plan.
4. **FR4**: The system shall categorize AI responses into diet types, workouts, meals, and health tips.
5. **FR5**: The system shall display responses in a readable and structured UI format.

**🔹 Non-Functional Requirements (NFR)**

1. **NFR1**: The system shall respond to queries within 3 seconds (excluding external API delay).
2. **NFR2**: The system shall maintain compatibility with modern web browsers (Chrome, Firefox).
3. **NFR3**: The application shall use environment variables to store sensitive API keys.
4. **NFR4**: The interface shall be mobile-responsive and lightweight.

**🔍 3. Feasibility Analysis**

| **Requirement ID** | **Type** | **Feasibility** |
| --- | --- | --- |
| FR1 | Functional | ✅ Feasible (API available and integrated) |
| FR2 | Functional | ✅ Feasible (HTML form exists) |
| FR3 | Functional | ✅ Feasible (Gemini integration tested) |
| FR4 | Functional | ✅ Feasible (response parser implemented) |
| FR5 | Functional | ✅ Feasible (Django templates in place) |
| NFR1 | Non-Functional | ⚠️ Depends on internet/API latency |
| NFR2 | Non-Functional | ✅ Confirmed via basic browser tests |
| NFR3 | Non-Functional | ✅ Used os.environ for keys |
| NFR4 | Non-Functional | ⚠️ Basic responsiveness, needs improvement |

**📋 4. Product Backlog**

| **ID** | **Feature Description** | **Priority** | **Notes** |
| --- | --- | --- | --- |
| PB1 | Food Nutrition Lookup using API | High | Already implemented |
| PB2 | User Form for Dietary and Fitness Inputs | High | Implemented |
| PB3 | AI Recommendation Engine Integration | High | Gemini used |
| PB4 | Result Parsing and Display Formatting | High | Implemented |
| PB5 | Responsive UI | Medium | Needs improvement |
| PB6 | Save Previous Searches (Session/DB-based) | Medium | Future enhancement |
| PB7 | User Account System (Authentication, Profiles) | Low | Future implementation |

**🗓️ 5. Sprint Backlog (Sprint 1 – Week 1)**

| **Task ID** | **Task Description** | **Assigned To** | **Status** |
| --- | --- | --- | --- |
| S1 | Setup Django Project & Static Templates | Developer | ✅ Done |
| S2 | Connect API Ninjas for nutrition data | Developer | ✅ Done |
| S3 | Implement data collection form | Developer | ✅ Done |
| S4 | Integrate Google Gemini API | Developer | ✅ Done |
| S5 | Display recommendations with parsing | Developer | ✅ Done |
| S6 | Draft initial documentation | Team Lead | ✅ Done |

**✅ Summary**

The project is **feasible** within scope and time constraints. Core APIs are functional, and the basic system is in place. Phase 2 focused on refining the requirements, verifying feasibility, and defining realistic deliverables via product/sprint backlogs.

**✅ Comprehensive Use Case Analysis – Meal & Calory Planner**

**🎭 Actors**

| **Actor** | **Description** |
| --- | --- |
| **User** | A visitor using the system to search food or get personalized plans |
| **Nutrition API** | External API for nutrition data (API Ninjas) |
| **Gemini AI Engine** | Google Generative AI used for generating health plans |
| **System** | Django web application processing user actions |
| **Admin (Future)** | Manages data, users, or settings in an extended version |

**📄 Use Case List**

| **Use Case Name** | **Description** |
| --- | --- |
| **Search Food Nutrition** | **User searches for food; system fetches nutritional details from API** |
| **View Nutrition Result** | **User views calculated nutrition and total score** |
| **Fill Recommendation Form** | **User fills form with dietary and fitness info** |
| **Generate Recommendations** | **System sends user data to Gemini and gets back plans** |
| **Display Recommendations** | **System parses and formats Gemini output** |
| **Print or Save Plan (Optional)** | **User saves/prints recommendation (to be implemented)** |
| **Manage System (Future Admin)** | **Admin manages data/models/settings (future)** |
| **Use Case Name** | **Description** |
| Submit Food Query | User enters a food name to retrieve nutritional details |
| Fetch Nutritional Data | System calls API Ninjas with the user's query |
| Display Nutritional Report | System displays food nutrient breakdown and total calories |
| Access Recommendation Page | User opens the diet & fitness recommendation form |
| Fill Health Profile Form | User inputs: dietary preferences, restrictions, health goals, etc. |
| Generate Prompt for Gemini | System builds a prompt with user input for Gemini API |
| Call Gemini API for Recommendations | System sends prompt to Gemini and fetches results |
| Parse Gemini AI Response | System processes response into: diets, workouts, meals, tips |
| Display Personalized Plan | Parsed content is shown in a structured format (HTML template) |
| Handle API Errors | System catches API errors and shows fallback messages |
| Access Admin Panel | (Future) Admin logs in to manage application settings |
| Edit Recommendations/Prompts | (Future) Admin fine-tunes Gemini prompt structure or templates |

**📘 Class Diagram Documentation**

**Project**: Meal & Calory Planner  
**Phase**: Object-Oriented Design  
**Focus**: Class Diagram — Attributes, Methods, Relationships

**🎯 Class Descriptions**

**1. User**

*(Conceptual – future enhancement for authentication and personalization)*  
**Attributes:**

* user\_id: int – Unique user identifier
* name: str – User's full name
* email: str – User's email address
* preferences: dict – Saved preferences (e.g., diet, fitness goals)

**Methods:**

* get\_history() – Retrieve past queries or plans
* save\_query(data) – Save a food search or recommendation
* update\_profile() – Modify user settings or preferences

**2. RecommendationForm**

**Attributes:**

* dietary\_preferences: str
* fitness\_goal: str
* lifestyle\_factor: str
* dietary\_restrictions: str
* health\_condition: str
* your\_query: str

**Methods:**

* get\_clean\_data() – (Planned) returns validated dictionary input

**3. FoodQueryHandler**

**Attributes:**

* api\_key: str – API key for Nutrition API
* query: str – Food item searched by the user

**Methods:**

* fetch\_data() – Sends HTTP request to Nutrition API
* parse\_data() – Parses API JSON response
* calculate\_total() – Computes total nutrition/calorie score

**4. NutritionResult**

**Attributes:**

* food\_data: dict – Parsed nutrient data
* total\_score: float – Total calculated value

**Methods:**

* get\_macro\_summary() – Extracts carb, fat, protein details

**5. GeminiEngine**

**Attributes:**

* api\_key: str – Key for accessing Gemini
* model\_name: str – e.g., "gemini-1.5-pro"

**Methods:**

* generate\_prompt(form) – Builds structured AI prompt
* send\_prompt(prompt) – Sends prompt to Gemini, returns response

**6. RecommendationParser**

**Attributes:**

* response\_text: str – Raw output from Gemini AI
* sections: dict – Parsed plan components (diet, meals, tips)

**Methods:**

* parse\_sections() – Breaks text into structured sections
* get\_section(name) – Retrieves list for a specific category

**7. HomeView *(Django view/controller)***

**Methods:**

* handle\_post(request) – Processes food query POST
* handle\_get(request) – Displays homepage form

**8. RecommendationView *(Django view/controller)***

**Methods:**

* handle\_form(request) – Collects form data, processes AI response

**🔗 Class Relationships & Cardinalities**

| **Class A** | **Relation Type** | **Class B** | **Cardinality** |
| --- | --- | --- | --- |
| User | fills/submits | RecommendationForm | 1 → 1 |
| User | queries | FoodQueryHandler | 1 → 1 |
| FoodQueryHandler | creates | NutritionResult | 1 → 1 |
| RecommendationForm | passed to | GeminiEngine | 1 → 1 |
| GeminiEngine | feeds into | RecommendationParser | 1 → 1 |
| RecommendationParser | returns suggestions to | User | 1 → 1 |

**🧠 UML Design Relationships**

| **Type** | **Present?** | **Example** |
| --- | --- | --- |
| Association | ✅ Yes | View uses Form, Handler |
| Dependency | ✅ Yes | RecommendationParser depends on GeminiEngine output format |
| Aggregation | ✅ Weak | RecommendationView aggregates Form, Engine, Parser |
| Composition | ❌ No | No tight object ownership or lifecycle binding |
| Generalization | ❌ No | No inheritance or abstract base classes used |

**📦 Object Instances (with Sample Data)**

**🧍 user1: User**

| **Attribute** | **Value** |
| --- | --- |
| user\_id | 101 |
| name | "Alice" |
| email | "alice@example.com" |
| preferences | { "diet": "vegan", "goal": "weight loss" } |

**📝 form1: RecommendationForm**

| **Attribute** | **Value** |
| --- | --- |
| dietary\_preferences | "Vegan" |
| fitness\_goal | "Weight Loss" |
| lifestyle\_factor | "Sedentary" |
| dietary\_restrictions | "Gluten-Free" |
| health\_condition | "Type 2 Diabetes" |
| your\_query | "Quick fat-burning meal?" |

**⚙️ engine1: GeminiEngine**

| **Attribute** | **Value** |
| --- | --- |
| model\_name | "gemini-1.5-pro" |
| api\_key | (hidden) |

**📊 parser1: RecommendationParser**

| **Attribute** | **Value** |
| --- | --- |
| response\_text | "1. Keto diet...\n2. Mediterranean diet...\n..." |
| sections |  |

json

CopierModifier

{

"diet\_types": [

"Keto diet",

"Mediterranean diet",

"Vegan diet",

"Low-carb diet",

"Paleo diet"

],

"workouts": [

"HIIT (High-Intensity Interval Training)",

"Yoga",

"Resistance training",

"Brisk walking",

"Swimming"

],

"breakfasts": [

"Avocado toast with gluten-free bread",

"Smoothie with spinach, banana, and protein powder",

"Oatmeal with almond milk and chia seeds",

"Vegan tofu scramble with veggies",

"Greek yogurt with berries (if allowed)"

],

"dinners": [

"Grilled salmon with quinoa and steamed broccoli",

"Stir-fried tofu with mixed vegetables",

"Lentil soup with gluten-free toast",

"Chicken breast with sweet potatoes and greens",

"Zucchini noodles with avocado pesto"

],

"additional\_tips": [

"Drink at least 2L of water daily",

"Include omega-3 supplements if not eating fish",

"Snack on nuts or protein bars",

"Avoid processed sugars",

"Prioritize sleep and reduce stress"

]

}

**🍌 query1: FoodQueryHandler**

| **Attribute** | **Value** |
| --- | --- |
| query | "banana" |
| api\_key | (hidden) |

**📈 result1: NutritionResult**

| **Attribute** | **Value** |
| --- | --- |
| food\_data | { "carbs": 22g, "fat": 0.3g, "sugar": 12g } |
| total\_score | 95 |

**🔗 Object Relationships**

| **From** | **To** | **Type** | **Description** |
| --- | --- | --- | --- |
| user1 | query1 | Association | Submits food query |
| query1 | result1 | Aggregation | Produces parsed result |
| user1 | form1 | Association | Fills recommendation form |
| form1 | engine1 | Dependency | Used to generate AI prompt |
| engine1 | parser1 | Association | Sends response to be parsed |
| parser1 | user1 | Association | Returns structured plan |

**📈 Runtime Flow (Snapshot Explanation)**

1. user1 inputs "banana" → query1 fetches API data → generates result1
2. user1 fills form1 with health data → passed to engine1
3. engine1 calls Gemini API → returns AI text
4. AI output parsed by parser1 → categories extracted
5. Recommendations shown to user1

Perfect! Here is a **detailed and professional textual explanation** of the **“Search Food Nutrition”** use case in sequence diagram format. This documentation is suitable for technical reports, academic submissions, and UML-based system analysis.

**📘 Sequence Diagram 1: Search Food Nutrition**

**✅ Use Case Description**

**🔹 Use Case Name:**

**Search Food Nutrition**

**🔹 Goal:**

Allow the user to enter a food item and receive a detailed nutritional breakdown including calories, carbohydrates, fats, and other macronutrients.

**🔹 Trigger:**

The user inputs a food item (e.g., “banana”) and submits the query via the web interface.

**🔹 Precondition:**

The Nutrition API (API Ninjas) must be reachable and functional, and the user must be on the homepage.

**🔹 Postcondition:**

The user receives a structured report showing the nutritional information of the queried food item.

**🧑‍🤝‍🧑 Actors Involved**

| **Actor** | **Description** |
| --- | --- |
| **User** | Person who initiates the search by submitting a food query |
| **Web Interface** | Front-end layer that captures the user’s input and renders output |
| **HomeView (Controller)** | Django view that handles the logic of processing the request |
| **FoodQueryHandler (Component)** | Class responsible for fetching and parsing data from the Nutrition API |
| **Nutrition API (External)** | Third-party API that provides nutrition data in JSON format |
| **NutritionResult (Object)** | Class used to store and compute the final displayed data |

**🔁 Sequence of Interactions**

| **Step** | **Sender** | **Receiver** | **Message/Method Call** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | User | Web Interface | Enters a food item (e.g., “banana”) and clicks submit | Input trigger |
| 2 | Web Interface | HomeView | POST / with query data | The front-end sends a form POST request with the query |
| 3 | HomeView | FoodQueryHandler | handler = FoodQueryHandler(query="banana") | Instantiate the class with the user’s query |
| 4 | FoodQueryHandler | Nutrition API | GET /v1/nutrition?query=banana with API key in headers | Sends request to external API to retrieve nutrition data |
| 5 | Nutrition API | FoodQueryHandler | JSON response with food nutrient data | Returns structured data (e.g., carbs, fats, sugar) |
| 6 | FoodQueryHandler | HomeView | parse\_data() and calculate\_total() | Parses the JSON and computes an overall nutrition score |
| 7 | HomeView | NutritionResult | result = NutritionResult(food\_data, total\_score) | Creates a result object to hold data |
| 8 | HomeView | Web Interface | Render home.html with result data | Passes nutrition and score data back to be displayed on the UI |
| 9 | Web Interface | User | Displays nutrient report | The user sees the breakdown and calorie count |

**📌 Notes:**

* Error handling (e.g., no result found, API failure) can be implemented inside the HomeView and reported back to the user with a fallback message.
* The NutritionResult class is an abstraction layer; it can also support additional methods like unit conversions or chart-ready formatting.

**🧠 Summary:**

* **Main Actor**: User
* **System Actors**: Web Interface, HomeView, FoodQueryHandler, Nutrition API
* **Output**: Structured nutrient report and total score
* **Technology Used**: Django (views), API Ninjas, HTML Template rendering

Excellent! Let's now proceed to the **second sequence diagram use case**:

**📘 Sequence Diagram 2: Generate Personalized Recommendations**

**✅ Use Case Description**

**🔹 Use Case Name:**

**Generate Personalized Diet and Workout Recommendations**

**🔹 Goal:**

Enable a user to receive personalized dietary and workout suggestions based on their health profile using the Google Gemini AI model.

**🔹 Trigger:**

The user navigates to the recommendation form page, fills in their profile, and submits the form.

**🔹 Precondition:**

* The form is loaded and accessible
* The Gemini AI API is configured and functional
* The user has valid input data

**🔹 Postcondition:**

The system returns a structured list of diet types, workouts, meals, and tips tailored to the user’s profile.

**🧑‍🤝‍🧑 Actors Involved**

| **Actor** | **Description** |
| --- | --- |
| **User** | The person requesting health recommendations |
| **Web Interface** | Front-end HTML form and result display |
| **RecommendationView** | Django view that processes the form and handles AI interaction |
| **RecommendationForm** | Object that holds and validates user input |
| **GeminiEngine** | Class that constructs prompts and communicates with Google Gemini AI |
| **RecommendationParser** | Processes and structures Gemini AI response into readable sections |

**🔁 Sequence of Interactions**

| **Step** | **Sender** | **Receiver** | **Message/Method Call** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | User | Web Interface | Fills and submits recommendation form | Inputs preferences, goals, health info |
| 2 | Web Interface | RecommendationView | POST /recommendation/ with form data | Form data sent to Django controller |
| 3 | RecommendationView | RecommendationForm | form = RecommendationForm(\*\*POST) | Form instantiated with submitted data |
| 4 | RecommendationView | GeminiEngine | generate\_prompt(form\_data) | Form data formatted into a detailed prompt |
| 5 | GeminiEngine | Google Gemini API | Send prompt via SDK (model.generate\_content(...)) | Gemini model receives the prompt |
| 6 | Google Gemini API | GeminiEngine | Returns plain-text response | AI returns suggestions for diet, workouts, meals, tips |
| 7 | GeminiEngine | RecommendationParser | parser = RecommendationParser(response\_text) | AI response is sent for parsing |
| 8 | RecommendationParser | RecommendationView | sections = parser.parse\_sections() | Extract structured lists from the text |
| 9 | RecommendationView | Web Interface | Render recomendation.html with parsed data | Data passed to template for user display |
| 10 | Web Interface | User | Structured recommendations displayed | User views categorized suggestions on-screen |

**📦 Example User Input (Form)**

* **Dietary Preferences**: Vegan
* **Fitness Goal**: Weight Loss
* **Lifestyle Factor**: Sedentary
* **Dietary Restrictions**: Gluten-Free
* **Health Condition**: Diabetes
* **Your Query**: “I want a quick fat-burning meal and workout combo”

**📨 Example AI Output (Plain Text Summary)**

1. Diet Recommendations:

- Vegan diet

- Keto diet

...

2. Workout Options:

- HIIT

- Yoga

...

3. Meal Suggestions:

Breakfast: oatmeal, smoothie

Dinner: tofu stir-fry, salmon quinoa bowl

4. Additional Tips:

- Hydrate, sleep, avoid sugar...

**🧠 Summary**

* **Primary Actor**: User
* **System Actors**: Web Interface, Form, Controller View, GeminiEngine, Parser
* **Technology Used**: Django, Google Gemini AI (gemini-1.5-pro), HTML templates
* **Output**: Categorized plan with diet, workouts, meals, and health tips

Excellent! Let’s proceed with the **third sequence diagram use case**, focusing on error handling — a critical part of any robust system.

**📘 Sequence Diagram 3: Handle API Errors Gracefully**

**✅ Use Case Description**

**🔹 Use Case Name:**

**Handle External API Errors Gracefully**

**🔹 Goal:**

Ensure that if a third-party API (Nutrition API or Gemini API) fails or returns no data, the system catches the error and provides a helpful fallback message to the user.

**🔹 Trigger:**

The user submits a food query or recommendation form, but the external API fails to respond correctly.

**🔹 Precondition:**

The user has submitted a request that depends on an external API.

**🔹 Postcondition:**

The system catches the error, logs it if needed, and shows a helpful message like “Oops! There was an error retrieving your data.”

**🧑‍🤝‍🧑 Actors Involved**

| **Actor** | **Description** |
| --- | --- |
| **User** | The person performing an action that leads to an API call |
| **Web Interface** | Sends request and receives final user-facing response |
| **Controller View** | Either HomeView or RecommendationView in Django |
| **API Client** | Could be FoodQueryHandler or GeminiEngine depending on the use case |
| **External API** | Failing system (Nutrition API or Gemini API) |

**🔁 Sequence of Interactions**

| **Step** | **Sender** | **Receiver** | **Message/Method Call** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | User | Web Interface | Submits query (food or recommendation form) | User initiates an API-dependent operation |
| 2 | Web Interface | Controller View | POST request sent to Django view | Sends the form/query data to the backend |
| 3 | Controller View | API Client | Call fetch\_data() or send\_prompt() | Tries to call Nutrition API or Gemini AI |
| 4 | API Client | External API | Request is sent | API call initiated |
| 5 | External API | API Client | **Error or Timeout** | API fails to return a valid response (e.g., 500, 404, timeout) |
| 6 | API Client | Controller View | Throws or returns error | Caught inside a try/except block |
| 7 | Controller View | Web Interface | Return fallback context | Renders HTML template with an error message |
| 8 | Web Interface | User | Shows message: “Oops! There was an error…” | User sees a polite and friendly error message instead of a crash |

**📌 Error Sources Handled**

* **Nutrition API**
  + Missing query
  + Network timeout
  + Invalid response format
* **Gemini AI**
  + Rate limit
  + Invalid prompt
  + API key expired

**🧠 Summary**

* **Primary Actor**: User
* **System Actors**: Web Interface, Django View, API Client
* **Outcome**: Graceful fallback with a user-friendly error message
* **Key Design**: Robust use of try/except, non-breaking UI, logging (optional)

Great! Let’s now cover the **fourth sequence diagram use case** which is a complete end-to-end flow — from form submission to parsing the AI response and rendering the structured output.

**📘 Sequence Diagram 4: Submit Recommendation Form and Parse Response**

**✅ Use Case Description**

**🔹 Use Case Name:**

**Submit Recommendation Form and Parse AI Response**

**🔹 Goal:**

Process the user's health and fitness input, generate a personalized recommendation using Gemini AI, parse the response, and return categorized results to the user.

**🔹 Trigger:**

The user submits the health profile form on the "Recommendations" page.

**🔹 Precondition:**

* The form is correctly loaded
* Gemini AI API is accessible
* All fields are filled appropriately

**🔹 Postcondition:**

A categorized output is displayed: diet types, workouts, meals, and health tips — extracted from the AI response.

**🧑‍🤝‍🧑 Actors Involved**

| **Actor** | **Description** |
| --- | --- |
| **User** | The person submitting their health data |
| **Web Interface** | HTML form for collecting user input and displaying results |
| **RecommendationView** | Django view that handles form processing and Gemini API interaction |
| **RecommendationForm** | Structure to capture and validate form data |
| **GeminiEngine** | Constructs prompt and communicates with the Gemini AI model |
| **RecommendationParser** | Extracts structured lists from the plain-text response |

**🔁 Sequence of Interactions**

| **Step** | **Sender** | **Receiver** | **Message/Method Call** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | User | Web Interface | Fills health form and submits it | Input includes diet, goal, restrictions, health status |
| 2 | Web Interface | RecommendationView | POST /recommendation/ | Sends data to Django backend |
| 3 | RecommendationView | RecommendationForm | form = RecommendationForm(\*\*POST) | Instantiate form with user input |
| 4 | RecommendationView | GeminiEngine | prompt = generate\_prompt(form\_data) | Builds structured prompt based on input |
| 5 | GeminiEngine | Gemini AI API | send\_prompt(prompt) | Sends prompt to Google Gemini model |
| 6 | Gemini AI API | GeminiEngine | Returns plain-text response | Example: Diets, workouts, meals, and tips |
| 7 | GeminiEngine | RecommendationParser | parser = RecommendationParser(response\_text) | Wraps response in parsing component |
| 8 | RecommendationParser | RecommendationView | sections = parse\_sections() | Extracts structured categories from response |
| 9 | RecommendationView | Web Interface | Render recomendation.html with parsed data | Data passed to HTML template |
| 10 | Web Interface | User | Display categorized recommendations | Diet types, workouts, meals, additional tips |

**📦 Categories Parsed from Gemini Response**

| **Category** | **Sample Output** |
| --- | --- |
| diet\_types | Keto, Vegan, Mediterranean, Low-carb, Paleo |
| workouts | Yoga, HIIT, Swimming, Strength Training, Brisk Walking |
| breakfasts | Oatmeal, Smoothies, Vegan Scramble |
| dinners | Grilled Salmon, Lentil Soup, Zucchini Noodles |
| additional\_tips | Drink 2L water, avoid sugar, take omega-3, sleep well |

**🧠 Summary**

* **Primary Actor**: User
* **Key System Components**: Web Interface, Django View, Gemini Engine, Parser
* **Main Flow**: Form → Prompt → Gemini → Plain Text → Parsed → Structured HTML Output
* **Success Outcome**: Categorized plan displayed to the user

Excellent! Let's complete the sequence diagram documentation set with the **fifth use case**, which combines both core flows: food lookup and AI-powered recommendations in one integrated scenario.

**📘 Sequence Diagram 5: Integrated Flow – Food Search + AI Recommendation**

**✅ Use Case Description**

**🔹 Use Case Name:**

**End-to-End System–API Integration Flow**

**🔹 Goal:**

Allow the user to search for a food item, view its nutritional content, and immediately proceed to receive personalized diet and workout suggestions — all in one session.

**🔹 Trigger:**

User lands on the homepage, searches for a food (e.g., “banana”), then proceeds to the recommendation section and submits their health profile.

**🔹 Precondition:**

* Nutrition API and Gemini AI are both accessible
* Both forms (food input and recommendation) are rendered properly
* Valid user input is submitted

**🔹 Postcondition:**

* The user receives a nutrition summary for the queried food
* The user also receives a structured recommendation plan (diets, workouts, meals, tips)

**🧑‍🤝‍🧑 Actors Involved**

| **Actor** | **Description** |
| --- | --- |
| **User** | Person performing the full food+health interaction |
| **Web Interface** | Front-end UI to manage form input/output |
| **HomeView** | Django view handling food queries and nutrition summary |
| **FoodQueryHandler** | Interacts with the Nutrition API to fetch data |
| **Nutrition API** | Third-party API providing food data |
| **RecommendationView** | Django view handling user health profile and AI generation |
| **GeminiEngine** | Constructs prompts and interacts with Gemini AI |
| **RecommendationParser** | Parses plain-text AI response into structured output |

**🔁 Sequence of Interactions**

| **Step** | **Sender** | **Receiver** | **Message/Method Call** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | User | Web Interface | Inputs food query (e.g., “banana”) | Starts nutrition lookup |
| 2 | Web Interface | HomeView | POST / with query | Sends query to controller |
| 3 | HomeView | FoodQueryHandler | fetch\_data() | API call to Nutrition API |
| 4 | FoodQueryHandler | Nutrition API | GET with query | Gets data for banana |
| 5 | Nutrition API | FoodQueryHandler | JSON response | Returns food nutrition data |
| 6 | FoodQueryHandler | HomeView | parse\_data(), calculate\_total() | Parses and computes score |
| 7 | HomeView | Web Interface | Render home.html with nutrition data | Nutrition result displayed |
| 8 | User | Web Interface | Fills recommendation form | Inputs health and fitness details |
| 9 | Web Interface | RecommendationView | POST /recommendation/ | Sends form to backend |
| 10 | RecommendationView | GeminiEngine | generate\_prompt(form\_data) | Formatted input → prompt |
| 11 | GeminiEngine | Gemini AI API | send\_prompt(prompt) | Sends prompt to Gemini |
| 12 | Gemini AI API | GeminiEngine | AI plain-text response | Returns recommendation content |
| 13 | GeminiEngine | RecommendationParser | Wrap response into parser | Passes plain-text for parsing |
| 14 | RecommendationParser | RecommendationView | parse\_sections() | Returns structured categories |
| 15 | RecommendationView | Web Interface | Render recomendation.html | Passes diet/workout/meal/tips to UI |
| 16 | Web Interface | User | User sees nutrition + full recommendation | Final structured result displayed |

**🧠 Summary**

* **Primary Actor**: User
* **Integrated Flow**:
  + Food Search → Nutrition API → Nutritional Summary
  + Health Form → Gemini AI → Personalized Plan
* **Technologies**: Django views, templates, API integrations (external), AI parsing
* **System Strength**: Shows end-to-end interactivity, API orchestration, and parsing flow