Project Goal:

* Create a Flask-based web application named "My-Health-Tracker".
* Provide a user-friendly interface for tracking daily calorie and macro-nutrient intake.
* Implement user-specific data storage using SQLite.
* Integrate with an LLM (OpenAI) and ChromaDB for food data lookup and storage.

**Core Features:**

1. **User Authentication:**
   * User registration (email, password, username).
   * User login.
   * Logout functionality.
   * Displaying the username on the top right corner of every page with an option to log out.
   * Track last login information.
2. **Record Meal:**
   * Text input for users to describe their meals (e.g., "I ate 250 grams boiled chicken").
   * LLM processing to extract food name, quantity, and units.
   * ChromaDB lookup for existing food data (calories, protein, fat, carbs).
   * LLM data retrieval for unknown food items.
   * Calculation of macro-nutrients based on quantity.
   * Displaying the nutrient information to the user for confirmation.
   * Error handling for insufficient user input.
   * Recording confirmed meals in the SQLite database (user-specific, date-specific).
   * Adding new food items to ChromaDB if they don't exist.
3. **Set Macro Limits:**
   * Allow users to set daily limits for calories, protein, fat, and carbs.
   * Input fields for each macro-nutrient and its limit.
   * Store user-specific limits in the SQLite database.
4. **View Today's Macros:**
   * Display today's consumed calories, protein, fat, and carbs.
   * Show remaining budget for each macro-nutrient.
   * Highlight exceeding limits in red.

**Technical Design:**

1. **Frontend (Flask/HTML/CSS/JavaScript):**
   * **Structure:**
     + A main layout template with the sidebar menu.
     + Individual templates for each page (record meal, set limits, view macros, login/register).
     + CSS for styling and responsiveness.
     + JavaScript for dynamic interactions (if needed).
   * **Sidebar Menu:**
     + Record Meal
     + Set Limits
     + View Today's Macros
     + Logout
   * **Top Right Corner:**
     + Display logged-in user's name.
     + Logout button.
2. **Backend (Flask/Python):**
   * **Flask App:**
     + Routes for each page and functionality.
     + Handle form submissions and database interactions.
   * **Database (SQLite):**
     + **registered\_users table:**
       - user\_id (primary key)
       - user\_email (unique)
       - password\_hash
       - user\_name
       - last\_login (timestamp)
     + **user\_calorie\_budget table:**
       - budget\_id (primary key)
       - user\_id (foreign key referencing registered\_users)
       - macro\_type (e.g., 'calories', 'protein', 'fat', 'carbs')
       - budget (numeric)
       - measurement\_type(e.g daily)
       - measurement\_unit(e.g. gram)
     + **user\_daily\_consumption\_records table:**
       - record\_id (primary key)
       - user\_id (foreign key referencing registered\_users)
       - date (date)
       - food\_name
       - quantity\_consumed
       - calories
       - protein
       - fat
       - carbs
   * **ChromaDB:**
     + A local vector database for food data.
     + Each entry will store:
       - food\_name
       - calories\_per\_100g
       - protein\_per\_100g
       - fat\_per\_100g
       - carbs\_per\_100g
   * **LLM Interaction:**
     + Parse meal descriptions to extract data.
     + Get food data if it's not in ChromaDB.
     + Calculate macro-nutrients.
3. **LLM Selection:**
   * **OpenAI:** You've chosen OpenAI, which is a solid choice. You'll need an API key for interacting with it.
   * You can specify the model like gpt-4-turbo or gpt-3.5-turbo based on your specific requirements.
   * The main prompt will be used to get macro nutrients for the weight of the food provided by the user.
   * The secondary prompt can be for extracting food, unit, and quantity from the user's input.

**Development Steps:**

1. **Setup:**
   * Create project directory.
   * Initialize virtual environment.
   * Install Flask, SQLite, ChromaDB, OpenAI libraries.
   * Install development dependencies as required.
2. **Database Design:**
   * Create the SQLite database schema.
   * Plan the ChromaDB structure.
3. **Backend:**
   * Build Flask routes and logic.
   * Database interactions.
   * LLM integrations.
4. **Frontend:**
   * Create HTML templates.
   * Add CSS styling.
   * Integrate with backend APIs.
5. **Testing:**
   * Unit tests for backend logic.
   * Manual testing for frontend.

**Enhancements and Suggestions:**

1. **User Profile:** Consider adding a user profile page where users can see their history and edit their details.
2. **Data Visualization:** Use charts and graphs to make the data more visually appealing.
3. **Mobile Responsiveness:** Ensure the app is responsive on different screen sizes.
4. **Error Handling:** Implement robust error handling for invalid inputs and API errors.
5. **Security:**
   * Hash passwords securely.
   * Protect against common web vulnerabilities.
6. **Advanced features:**
   * You can make food history also user linked.
   * There can be multiple measurement types like 'per-serving' etc.
   * You can extend to give ability to select meal time and to view history etc.

**Next Steps:**

Now that you have this comprehensive plan, I recommend we start with the following:

1. **Project Setup:** Create the project directory and initialize a virtual environment.
2. **Dependency Installation:** Install the necessary Python packages.
3. **Database setup**: Let's create the sqlite database, tables and connect to flask app.
4. **Folder Structure:** Let's create initial folder structure to put our code.

Amendment 1:

* **ChromaDB Entry:**
  + food\_name (string): The name of the food item (e.g., "boiled chicken", "bread", "apple").
  + measurement\_type (string): This will indicate whether the measurement is based on weight or units. It can be one of the following:
    - weight (e.g., grams, kilograms)
    - units (e.g., slices, pieces, cups)
  + measurement\_unit (string): The specific unit of measurement. Examples:
    - grams
    - slices
    - cups
    - pieces
    - each
    - kg
  + calories (numeric): Calories for the specified measurement unit.
  + protein (numeric): Protein content for the specified measurement unit.
  + fat (numeric): Fat content for the specified measurement unit.
  + carbs (numeric): Carbohydrate content for the specified measurement unit.

**Example Entries:**

* **Entry 1 (Boiled Chicken):**
  + food\_name: "boiled chicken"
  + measurement\_type: "weight"
  + measurement\_unit: "grams"
  + calories: 165.0 (per 100 grams)
  + protein: 31.0 (per 100 grams)
  + fat: 3.6 (per 100 grams)
  + carbs: 0.0 (per 100 grams)
* **Entry 2 (Bread):**
  + food\_name: "bread"
  + measurement\_type: "units"
  + measurement\_unit: "slices"
  + calories: 75.0 (per slice)
  + protein: 3.0 (per slice)
  + fat: 1.0 (per slice)
  + carbs: 13.0 (per slice)
* **Entry 3 (Apple):**
  + food\_name: "apple"
  + measurement\_type: "units"
  + measurement\_unit: "each"
  + calories: 95.0 (per apple)
  + protein: 0.5 (per apple)
  + fat: 0.3 (per apple)
  + carbs: 25.0 (per apple)

**Revised SQLite Database Design:**

The user\_daily\_consumption\_records table will also need to include these changes:

* **user\_daily\_consumption\_records table:**
  + record\_id (primary key)
  + user\_id (foreign key referencing registered\_users)
  + date (date)
  + food\_name
  + quantity\_consumed
  + measurement\_type (e.g weight, units)
  + measurement\_unit (e.g. gram, slices, each etc)
  + calories
  + protein
  + fat
  + carbs

**Logic Adjustments:**

1. **Meal Parsing:**
   * When the user inputs their meal, the LLM will need to identify not only the food name and quantity but also the implied measurement\_unit. For example, "I ate two slices of bread" implies a measurement unit of "slices", measurement type of units.
2. **ChromaDB Lookup:**
   * When looking up data in ChromaDB, we'll need to consider both the food\_name and the measurement\_unit and measurement\_type to find the correct entry.
3. **LLM Data Retrieval:**
   * If a food item isn't in ChromaDB, we'll need to ask the LLM for data that includes the measurement\_unit in addition to the food name and measurement\_type. For example, "What are the calories, protein, fat, and carbs in one slice of bread?"
4. **Calculation:**
   * The macro-nutrient calculation will now depend on both quantity\_consumed and the measurement\_unit. If the measurement\_type is 'units', the calculation will be based on the number of slices/pieces/cups etc.
   * If the measurement type is 'weight' we would need to do a division by 100 if chroma db had per 100 gram value and then multiply by the gram user provided.

**Updated Development Steps:**

1. **Setup:** (Same as before)
2. **Database Design:** (Revised)
   * Create the SQLite database schema with the updated user\_daily\_consumption\_records table.
   * Plan the ChromaDB structure with the new fields.
3. **Backend:** (Revised)
   * Modify Flask routes and logic to handle the new measurement fields.
   * Update database interactions to include the measurement\_unit.
   * Adjust LLM integrations to use the new prompts.
4. **Frontend:** (Revised)
   * Make sure the UI can handle cases for units and weight.
5. **Testing:** (Revised)
   * Create more comprehensive tests to cover both weight-based and unit-based measurements.

**Revised Next Steps:**

Since we've refined the database design, let's proceed with these actions:

1. **Project Setup:** Create the project directory and initialize a virtual environment.
2. **Dependency Installation:** Install the necessary Python packages.
3. **Database setup**: Let's create the sqlite database, tables and connect to flask app.
4. **Folder Structure:** Let's create initial folder structure to put our code.