Diet Manager

Date : 07-04-2025
Team members : Kalakuntla Rishika (2023101130), Sama Harshika (2023101004)

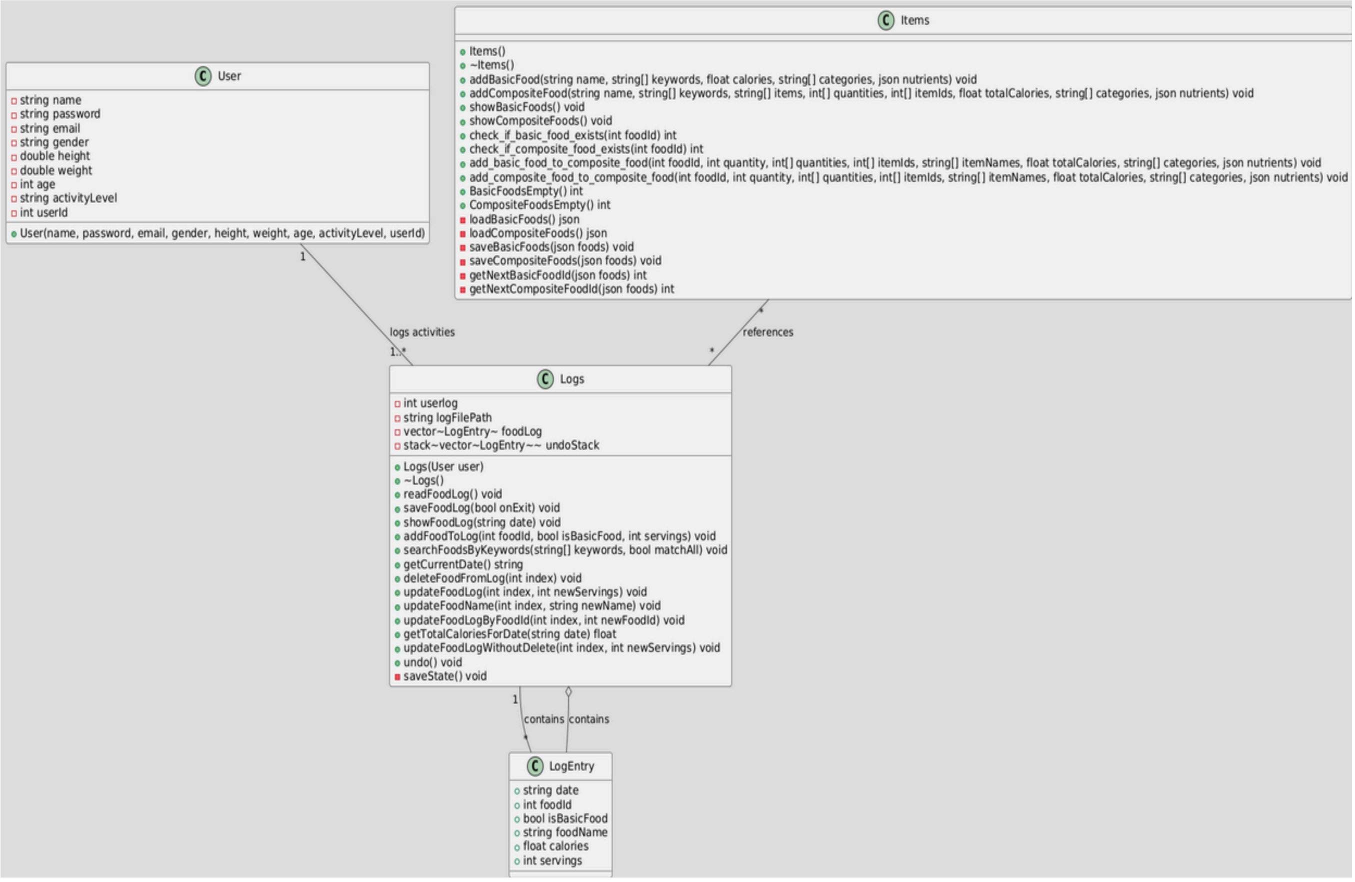
Product Overview:

YADA is a comprehensive diet management system designed to help users track their food intake, monitor the calorie consumption and achieve their dietary goals. The application maintains detailed food records for each user vehicle providing personalized calorie targets based on the user profiles.

Key Features:

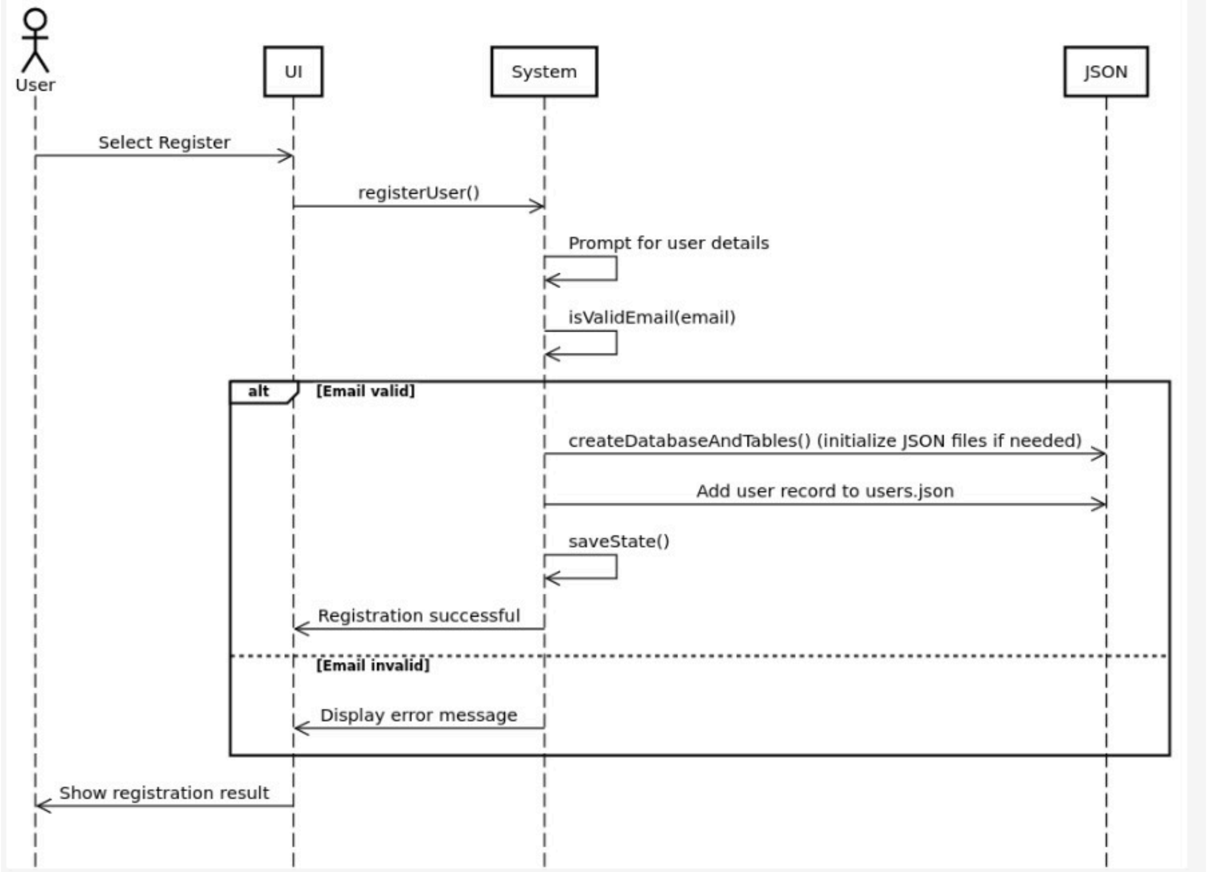
- **User Registration and Login** : Secure user registration and login system with comprehensive profile management.
- **Food Database Management :** Stores basic foods with identifiers, search keywords and calorie content.
- **Composite Food Creation :** Enable users to build composite foods from basic ingredients.
- **Daily Food Logging :** Maintains logs of consumed foods with date tracking and serving sizes.
- **Keyword Search Functionality :** Allows food selection through keyword searches with “all” or “any” match options.
- **User Profile & Diet Goals :** Records user’s gender, height, age, weight and activity level to calculate target calorie intake.
- **Multiple Calculation Methods :** Implements at least two (three) for computing daily calorie targets.
- **Unlimited Undo Capability :** Supports command reversal to an indefinite depth within sessions.
- **Date Selection :** Permits viewing and updating food information for any recorded date.

UML:

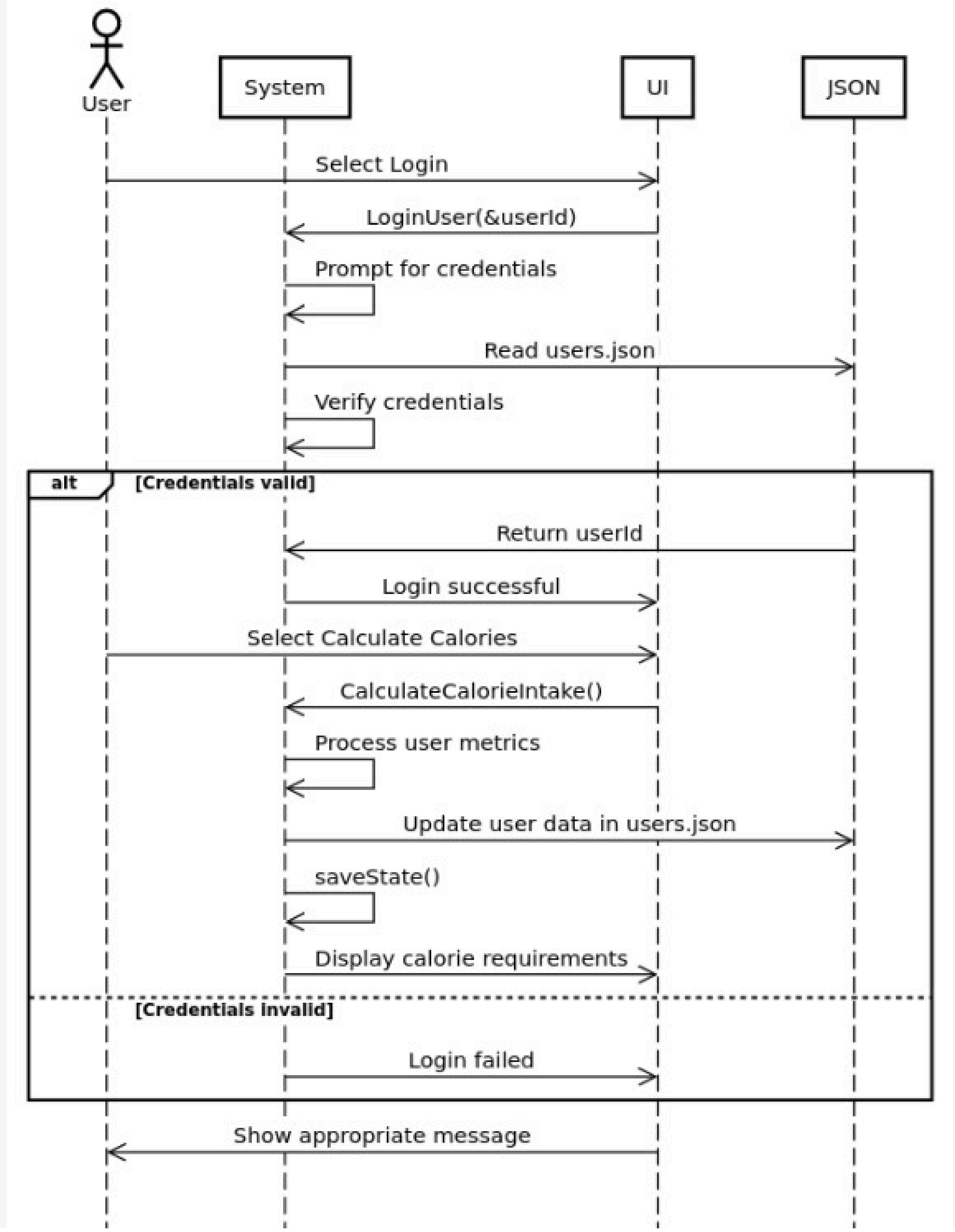

Items
Items()
~Items()
addBasicFood(string name, string[] keywords, float calories, string[] categories, json nutrients) void
addCompositeFood(string name, string[] keywords, string[] items, int[] quantities, int[] itemIds, float totalCalories, string[] categories, json nutrients) void
showBasicFoods() void
showCompositeFoods() void
check_if_basic_food_exists(int foodId) int
check_if_composite_food_exists(int foodId) int
add_basic_food_to_composite_food(int foodId, int quantity, int[] quantities, int[] itemIds, string[] itemNames, float totalCalories, string[] categories, json nutrients) void
add_composite_food_to_composite_food(int foodId, int quantity, int[] quantities, int[] itemIds, string[] itemNames, float totalCalories, string[] categories, json nutrients) void
BasicFoodsEmpty() int
CompositeFoodsEmpty() int
loadBasicFoods() json
loadCompositeFoods() json
saveBasicFoods(json foods) void
saveCompositeFoods(json foods) void
getNextBasicFoodId(json foods) int
getNextCompositeFoodId(json foods) int
User
string name
string password
string email
string gender
double height
double weight
int age
string activityLevel
int userId
User(name, password, email, gender, height, weight, age, activityLevel, userId)
1
logs activities
1..*
*
references
*
Logs
int userlog
string logFilePath
vector~LogEntry~ foodLog
stack~vector~LogEntry~~ undoStack
Logs(User user)
~Logs()
readFoodLog() void
saveFoodLog(bool onExit) void
showFoodLog(string date) void
addFoodToLog(int foodId, bool isBasicFood, int servings) void
searchFoodsByKeywords(string[] keywords, bool matchAll) void
getCurrentDate() string
deleteFoodFromLog(int index) void
updateFoodLog(int index, int newServings) void
updateFoodName(int index, string newName) void
updateFoodLogByFoodId(int index, int newFoodId) void
getTotalCaloriesForDate(string date) float
updateFoodLogWithoutDelete(int index, int newServings) void
undo() void
saveState() void
1
contains
contains
*
LogEntry
string date
int foodId
bool isBasicFood
string foodName
float calories
int servings

Sequence Diagrams:

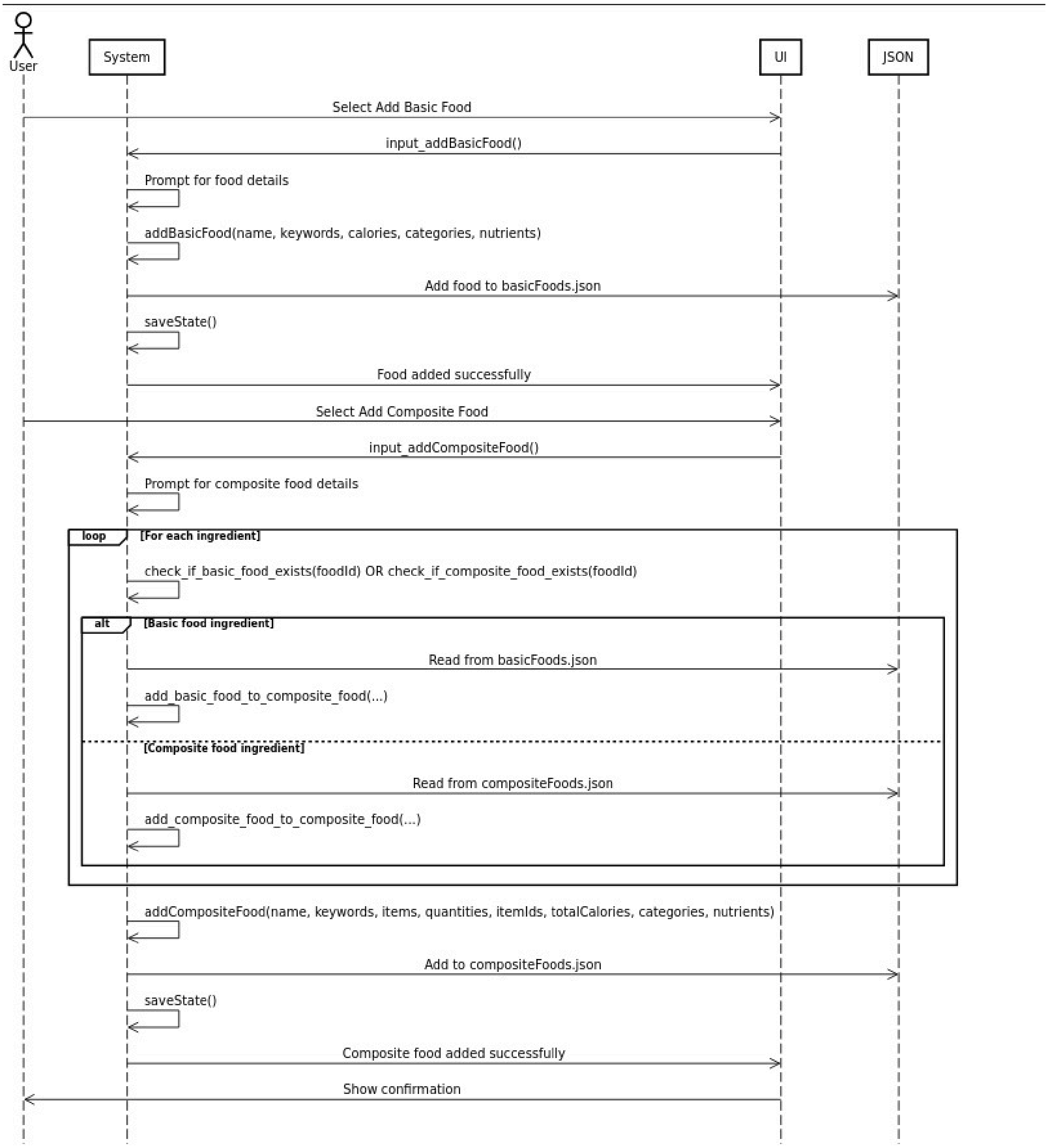
1. User Registration Sequence Diagram



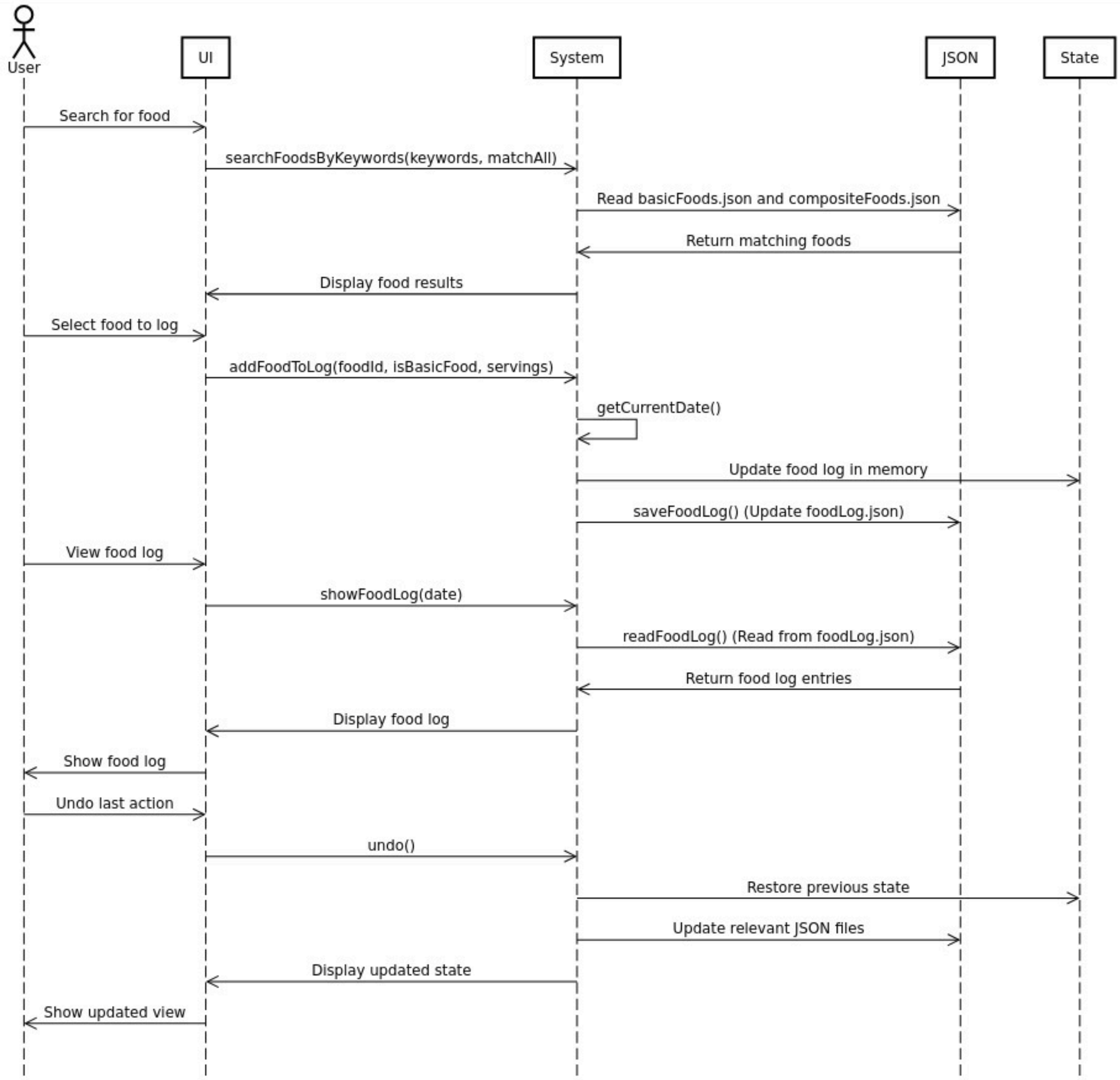
2. User Login and Calorie Calculation sequence Diagram



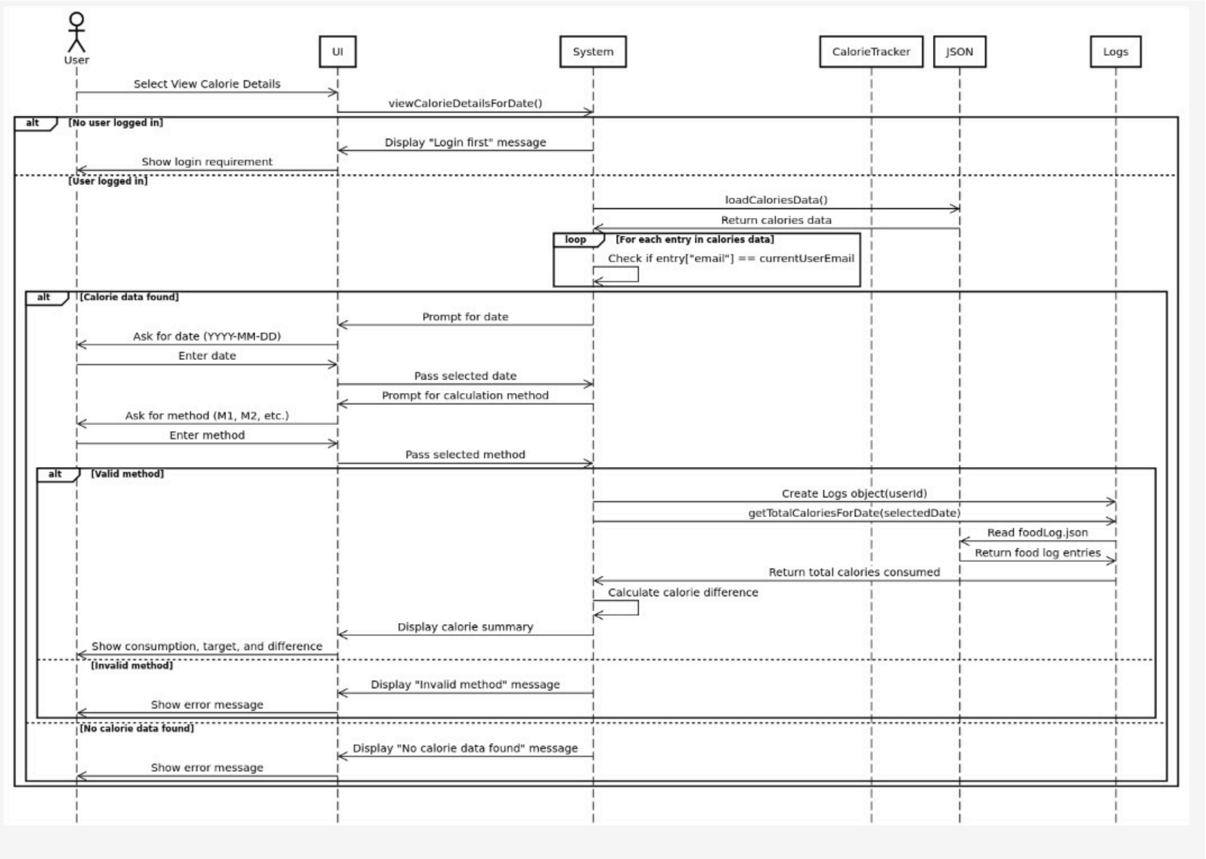
3. Food Management Sequence Diagram


User
System
UI
JSON
Select Add Basic Food
input_addBasicFood()
Prompt for food details
addBasicFood(name, keywords, calories, categories, nutrients)
Add food to basicFoods.json
saveState()
Food added successfully
Select Add Composite Food
input_addCompositeFood()
Prompt for composite food details
loop
[For each ingredient]
check_if_basic_food_exists(foodId) OR check_if_composite_food_exists(foodId)
alt
[Basic food ingredient]
Read from basicFoods.json
add_basic_food_to_composite_food(...)
[Composite food ingredient]
Read from compositeFoods.json
add_composite_food_to_composite_food(...)
addCompositeFood(name, keywords, items, quantities, itemIds, totalCalories, categories, nutrients)
Add to compositeFoods.json
saveState()
Composite food added successfully
Show confirmation

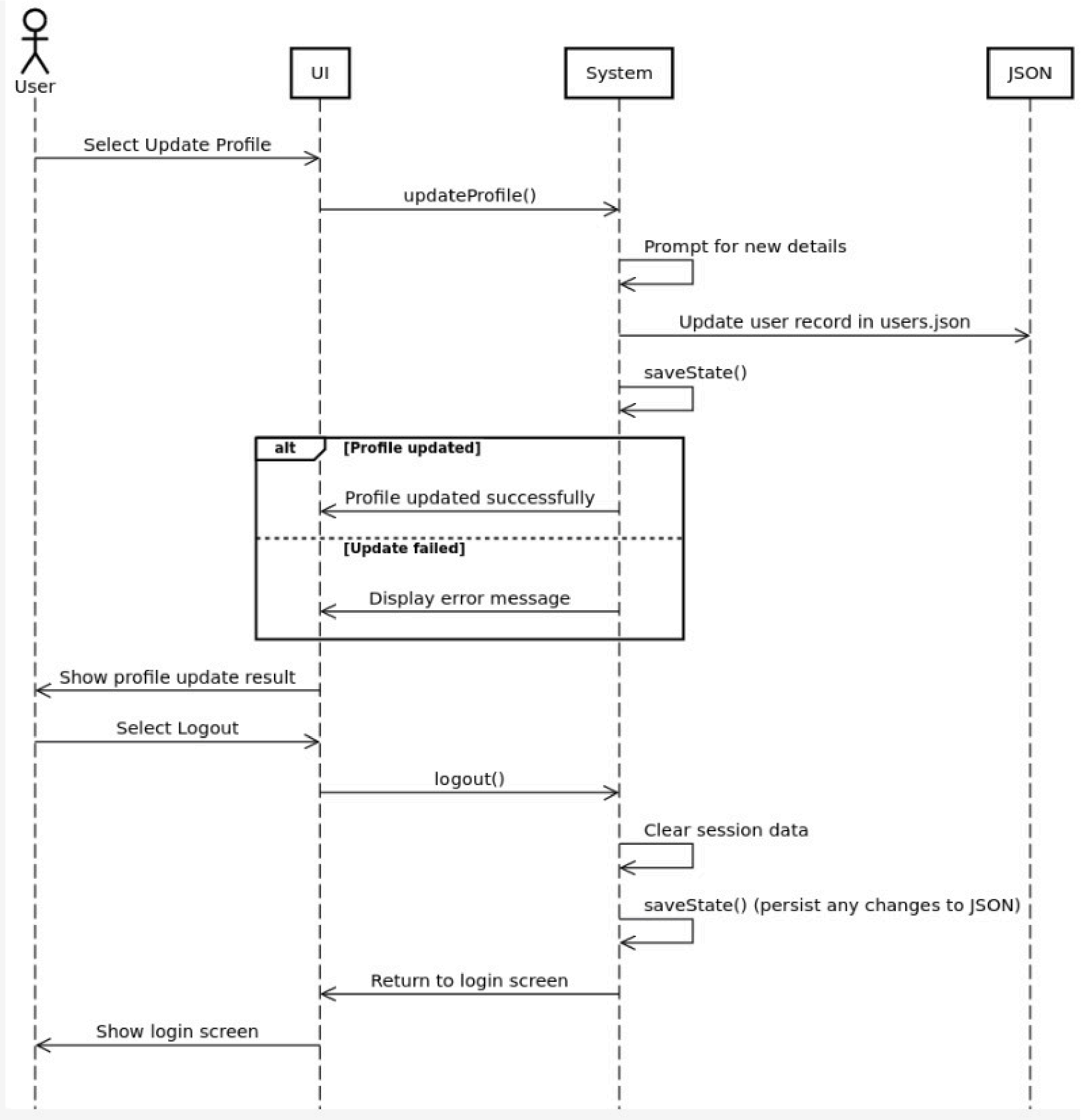
4. Food Logging sequence Diagram


User
UI
System
JSON
State
Search for food
searchFoodsByKeywords(keywords, matchAll)
Read basicFoods.json and compositeFoods.json
Return matching foods
Display food results
Select food to log
addFoodToLog(foodId, isBasicFood, servings)
getCurrentDate()
Update food log in memory
saveFoodLog() (Update foodLog.json)
View food log
showFoodLog(date)
readFoodLog() (Read from foodLog.json)
Return food log entries
Display food log
Show food log
Undo last action
undo()
Restore previous state
Update relevant JSON files
Display updated state
Show updated view

5. Viewing calorie Sequence Diagram


User
UI
System
CalorieTracker
JSON
Logs
Select View Calorie Details
viewCalorieDetailsForDate()
alt
[No user logged in]
Display "Login first" message
Show login requirement
[User logged in]
loadCaloriesData()
Return calories data
loop
[For each entry in calories data]
Check if entry["email"] == currentUserEmail
alt
[Calorie data found]
Prompt for date
Ask for date (YYYY-MM-DD)
Enter date
Pass selected date
Prompt for calculation method
Ask for method (M1, M2, etc.)
Enter method
Pass selected method
alt
[Valid method]
Create Logs object(userId)
getTotalCaloriesForDate(selectedDate)
Read foodLog.json
Return food log entries
Return total calories consumed
Calculate calorie difference
Display calorie summary
Show consumption, target, and difference
[Invalid method]
Display "Invalid method" message
Show error message
[No calorie data found]
Display "No calorie data found" message
Show error message

6. User Profile Management Sequence Diagram



Design Reflects:

- **Low Coupling:**
  - Low coupling ensures that components are minimally dependent on each other, making the system easier to modify and maintain.
  - Classes like Users, Items and Logs are self contained and handle specific responsibilities like food management, logging and user management.
  - Json file operations are encapsulated within the Items class ensuring that the other components do not directly interact with file I/O.
  - The Logs class manages food logs independently of the Items and User classes, reducing interdependencies.
- **High Cohesion:**
  - High ensures that each class or module has a single, well defined responsibility.
  - The Items class focuses on managing the basic and composite foodItems including adding, retrieving and validating the items.
  - The Logs class is responsible for managing user specific food logs, including adding entries, undoing and saving the actions.
  - The User class handles user related operations such as registration, login and profile updates.
  - Each method within these classes has a clear and specific purpose such as addBasicFood for adding a new basic food item or updateFoodLog for modifying the food log entry.

- **Separation of Concerns:**
  - Separation of concerns ensures that the different aspects of the systems are handled by distinct modules or classes.
  - The system separates the concerns into distinct classes:
    - Items handles food-related operations.
    - Logs manages food logs and user-specific data.
    - User handles user registration, login and profile updates.
  - The main.cpp file acts as entry point and orchestrates the interactions between these classes without embedding their logic.

- **Information Hiding:**
  - Information hiding ensures that internal details of the class are not exposed promoting encapsulation.
  - Internal data structures and helper methods like loadBasicFoods and getNextBasicFoodId are private within Items class ensuring that the external components cannot directly access and modify them.
  - The Logs class encapsulates the food log data and provides public methods like addFoodToLog and showFoodLog has control access.
  - The User class hides user details and provides methods like registerUser and updateProfile to interact with user data.

- **Law of Demeter:**
  - The law of demeter ensures that a method only interacts with its immediate collaborators, reducing the dependencies.

    - Methods in the Items class interact only with their direct collaborators such as JSON files and internal data structures rather than relying on the other classes.
    - The Logs class interacts with its internal foodLog vector and file I/O operations but doesn't directly manipulate Items or User objects.
    - The main.cpp file coordinates the high level operations without directly accessing the internals of the Items, Logs or the User classes.

- **Extensibility and Maintainability:**
    - The system should be easy to extend and maintain overtime.
    - The modular design allows new features like additional food attributes or new calorie calculation methods to be added without affecting existing functionality.
    - Helper methods like loadBasicFoods and saveBasicFoods centralize the file operations making it easier to update the storage format if needed.
    - The use of structured exception handling ensures that the errors are gracefully handled improving the robustness.

- **Reusability:**
    - Reusability ensures that components can be reused across different parts of the system.
    - The Items class provides reusable methods like addBasicFood and showBasicFoods which can be called independently in different contexts.
    - The Logs class can manage food logs for any user as it dynamically loads the appropriate log file based on their user ID.
    - The User class can handle multiple users without requiring changes to its implementation.

- **Scalability:**
    - Scalability ensures that the system can handle increased data or user load without significant changes.
    - JSON files are used for data storage, allowing the systems to scale by simply adding more entries without modifying the code.
    - The modular design ensures that the new food items, users or log entires are added without affecting the overall structure.

- **Readability and Simplicity:**
    - Readable and simple code is easier to understand and maintain.
    - Clear method names and structured code improve readability.
    - The use of comments and consistent formatting makes the code easier to follow.

Stronger and Weaker aspects:

Strongest aspects :

- **Separation of concerns:**
    - The design effectively separates responsibilities across different classes Items, Logs and User.
    - Each class handles a specific domain like Food Management, User Management and Logging respectively making the system modular and easier to maintain or extend.
- **Extensibility:**
    - The use of JSON files of storing data like basic_foods.json, composite_foods.json, users.json, calories.json allows the system to scale easily.
    - Adding new features such as additional food attributes or new calorie calculation methods can be done with minimal changes to the existing code.

Weakest aspects :

- **Redundant and Inefficient data handling:**
    - The design repeatedly loads and parses JSON files, for operations like searching, updating or adding entities.
    - This approach is inefficient especially for larger datasets, as it involves unnecessary file I/O and parsing overhead.

- **Inconsistent error handling:**
    - Many functions print error message directly to the terminal rather than returning error information.