Design Documentation

TITLE: YADA (Yet Another Diet App)

DATE: 06-05-2025

TEAM MEMBERS:

- 2023101044 - Sai charan Bakaram
- 2023101036 - Mohith Tondepu

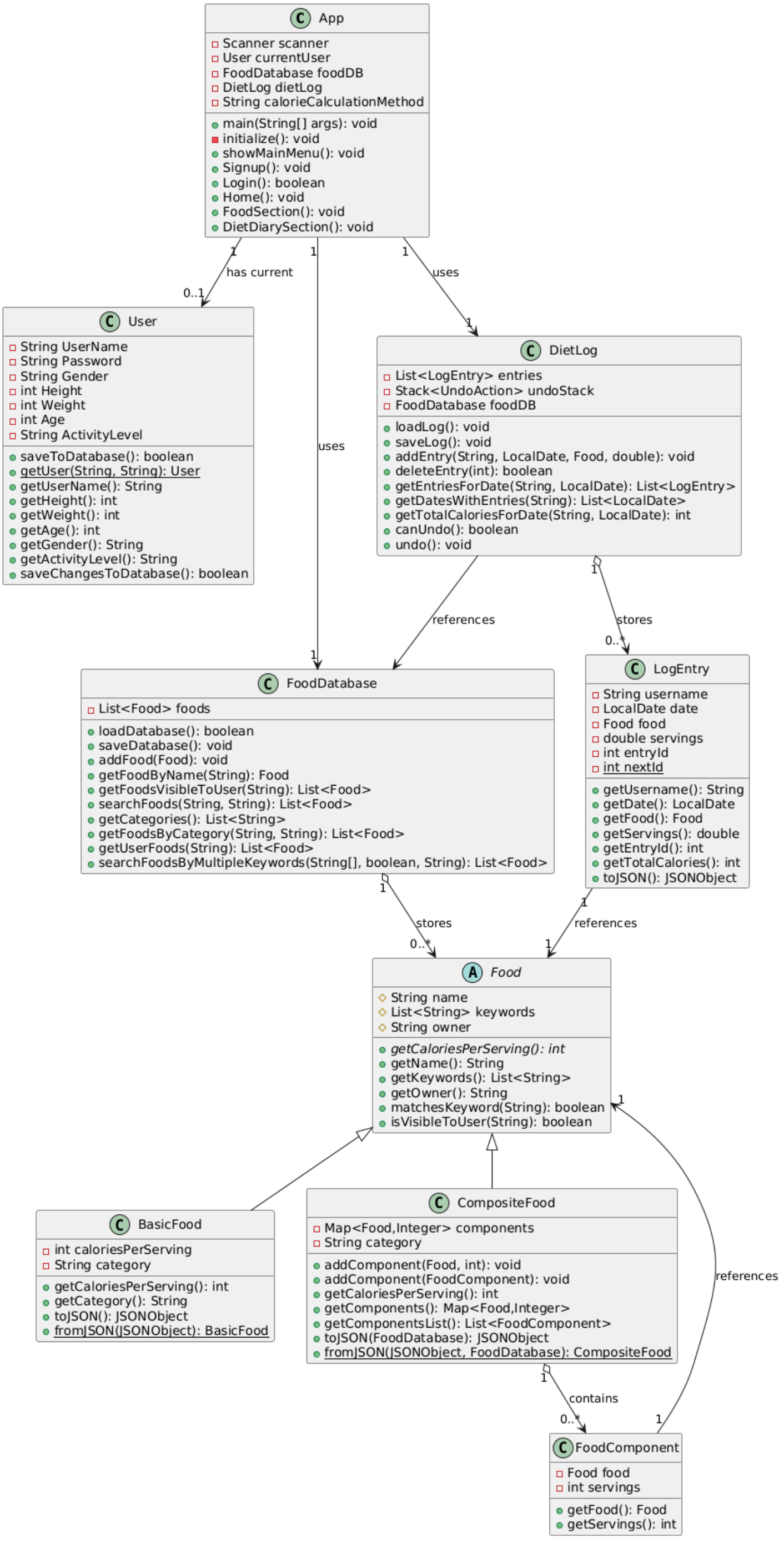
Overview

YADA is a diet tracking application designed to help users monitor their food consumption and achieve their nutritional goals. The application allows users to track daily food intake, search a comprehensive food database, create custom foods, calculate nutritional needs based on personal data.

Key features include:

- User account management with personal profile data and diet goal data
- Comprehensive food database with search capabilities
- Creation of custom basic and composite (recipe) foods
- Daily food logging with serving size tracking
- Nutritional needs calculator based on user profile
- Automatic update of servings when adding the same food multiple times in a day

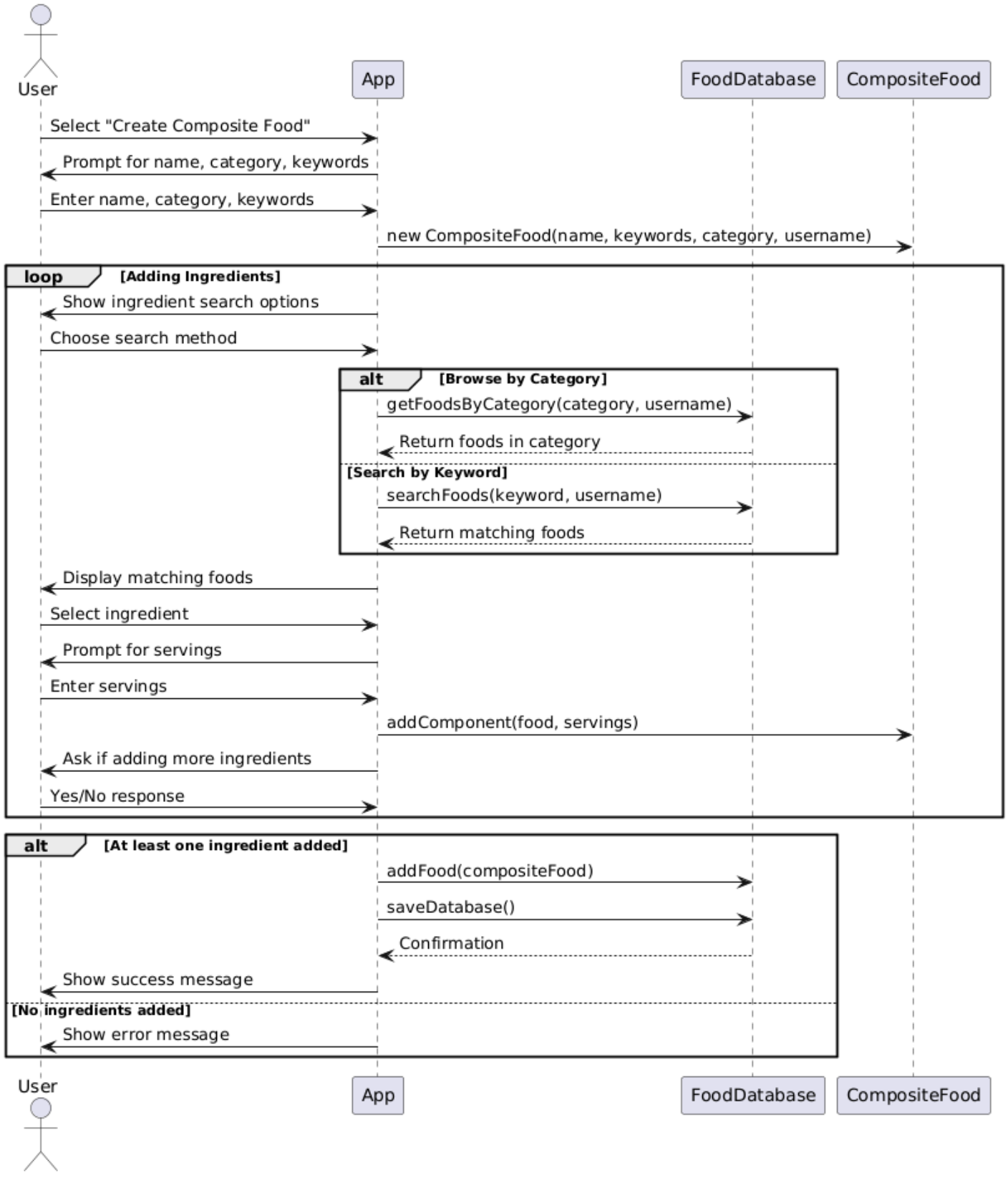
UML CLASS DIAGRAM


App
- Scanner scanner
- User currentUser
- FoodDatabase foodDB
- DietLog dietLog
- String calorieCalculationMethod
+ main(String[] args): void
- initialize(): void
+ showMainMenu(): void
+ Signup(): void
+ Login(): boolean
+ Home(): void
+ FoodSection(): void
+ DietDiarySection(): void
1
has current
0..1
1
uses
1
1
uses
User
- String UserName
- String Password
- String Gender
- int Height
- int Weight
- int Age
- String ActivityLevel
+ saveToDatabase(): boolean
+ getUser(String, String): User
+ getUserName(): String
+ getHeight(): int
+ getWeight(): int
+ getAge(): int
+ getGender(): String
+ getActivityLevel(): String
+ saveChangesToDatabase(): boolean
DietLog
- List<LogEntry> entries
- Stack<UndoAction> undoStack
- FoodDatabase foodDB
+ loadLog(): void
+ saveLog(): void
+ addEntry(String, LocalDate, Food, double): void
+ deleteEntry(int): boolean
+ getEntriesForDate(String, LocalDate): List<LogEntry>
+ getDatesWithEntries(String): List<LocalDate>
+ getTotalCaloriesForDate(String, LocalDate): int
+ canUndo(): boolean
+ undo(): void
1
references
1
stores
0..*
1
FoodDatabase
- List<Food> foods
+ loadDatabase(): boolean
+ saveDatabase(): void
+ addFood(Food): void
+ getFoodByName(String): Food
+ getFoodsVisibleToUser(String): List<Food>
+ searchFoods(String, String): List<Food>
+ getCategories(): List<String>
+ getFoodsByCategory(String, String): List<Food>
+ getUserFoods(String): List<Food>
+ searchFoodsByMultipleKeywords(String[], boolean, String): List<Food>
LogEntry
- String username
- LocalDate date
- Food food
- double servings
- int entryId
- int nextId
+ getUsername(): String
+ getDate(): LocalDate
+ getFood(): Food
+ getServings(): double
+ getEntryId(): int
+ getTotalCalories(): int
+ toJSON(): JSONObject
1
stores
0..*
1
references
Food
String name
List<String> keywords
String owner
+ getCaloriesPerServing(): int
+ getName(): String
+ getKeywords(): List<String>
+ getOwner(): String
+ matchesKeyword(String): boolean
+ isVisibleToUser(String): boolean
1
references
BasicFood
- int caloriesPerServing
- String category
+ getCaloriesPerServing(): int
+ getCategory(): String
+ toJSON(): JSONObject
+ fromJSON(JSONObject): BasicFood
CompositeFood
- Map<Food,Integer> components
- String category
+ addComponent(Food, int): void
+ addComponent(FoodComponent): void
+ getCaloriesPerServing(): int
+ getComponents(): Map<Food,Integer>
+ getComponentsList(): List<FoodComponent>
+ toJSON(FoodDatabase): JSONObject
+ fromJSON(JSONObject, FoodDatabase): CompositeFood
1
contains
0..*
1
FoodComponent
- Food food
- int servings
+ getFood(): Food
+ getServings(): int

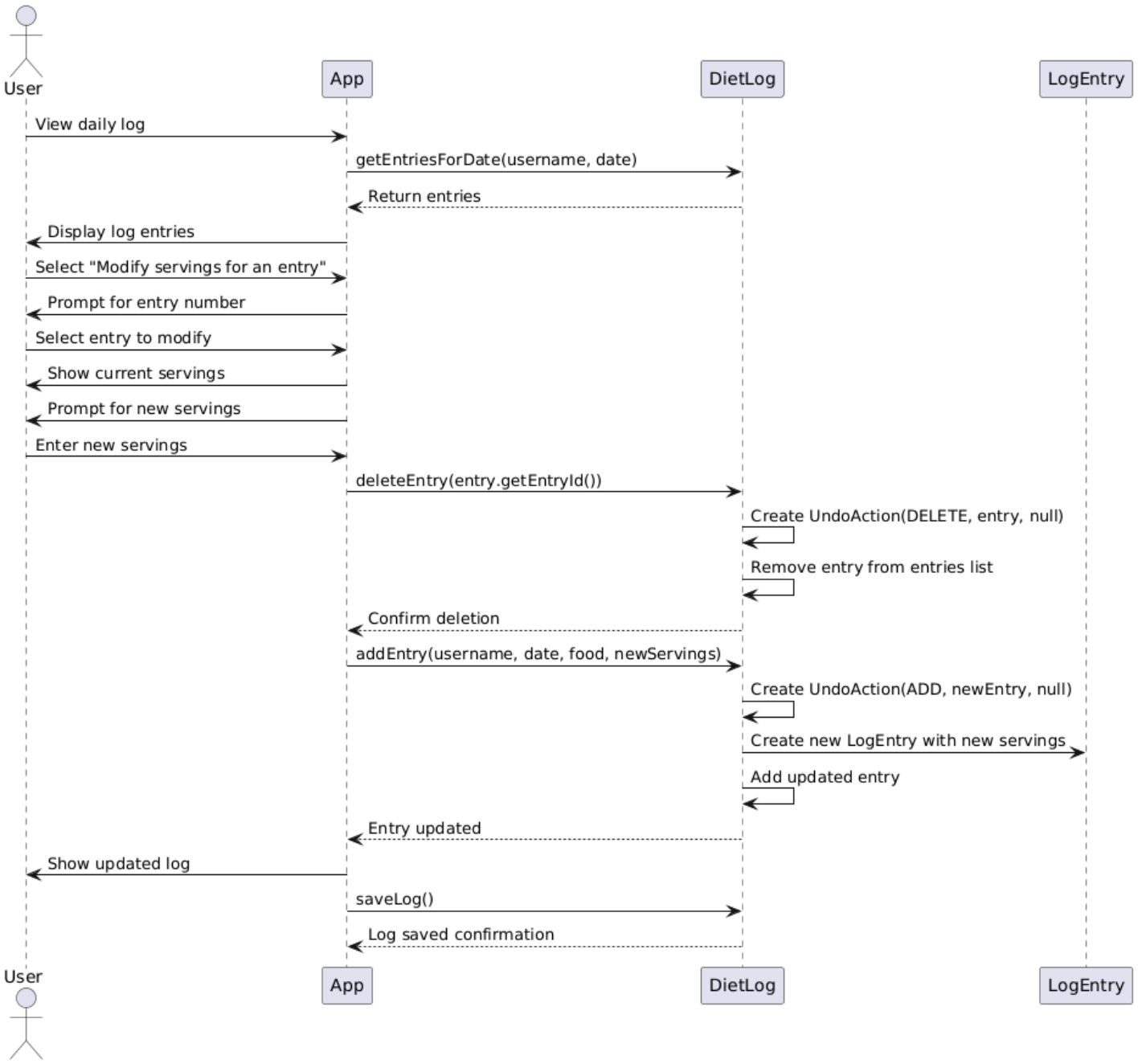
SEQUENCE DIAGRAMS:

FOOD LOGGING

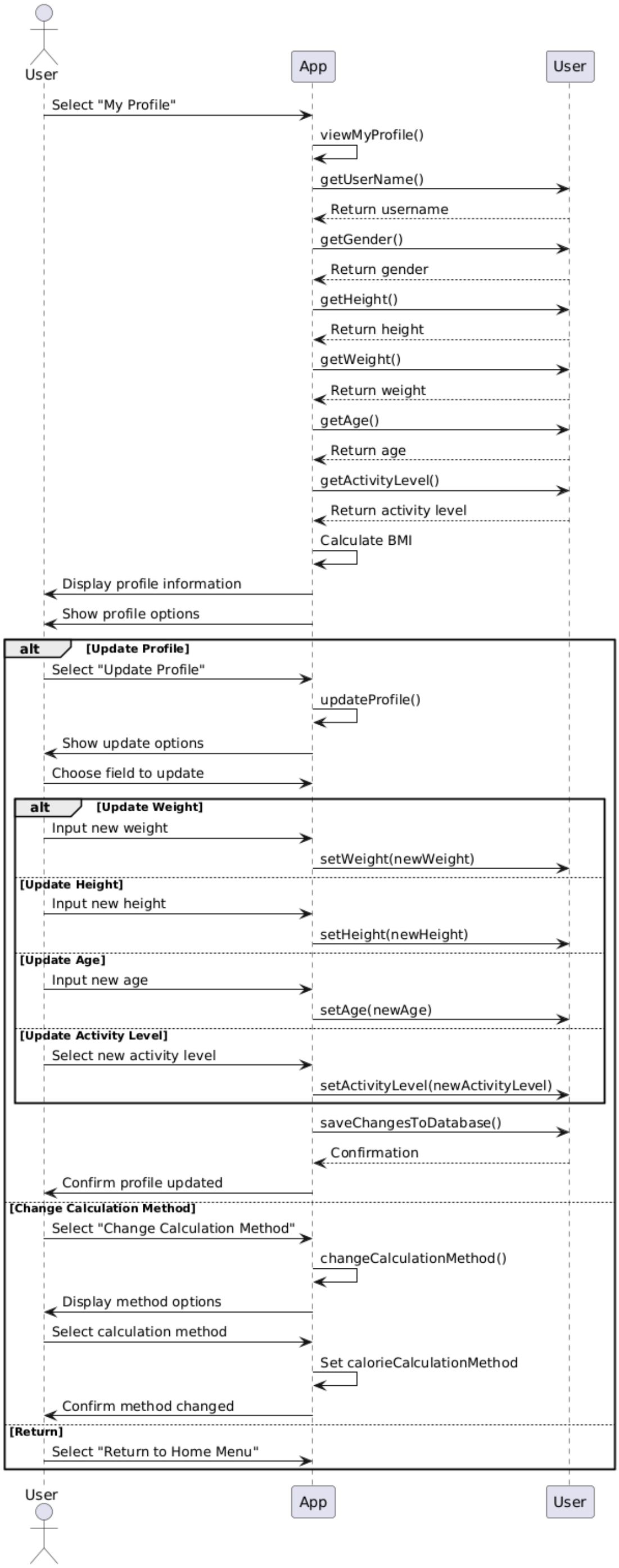
CREATE COMPOSITE FOOD



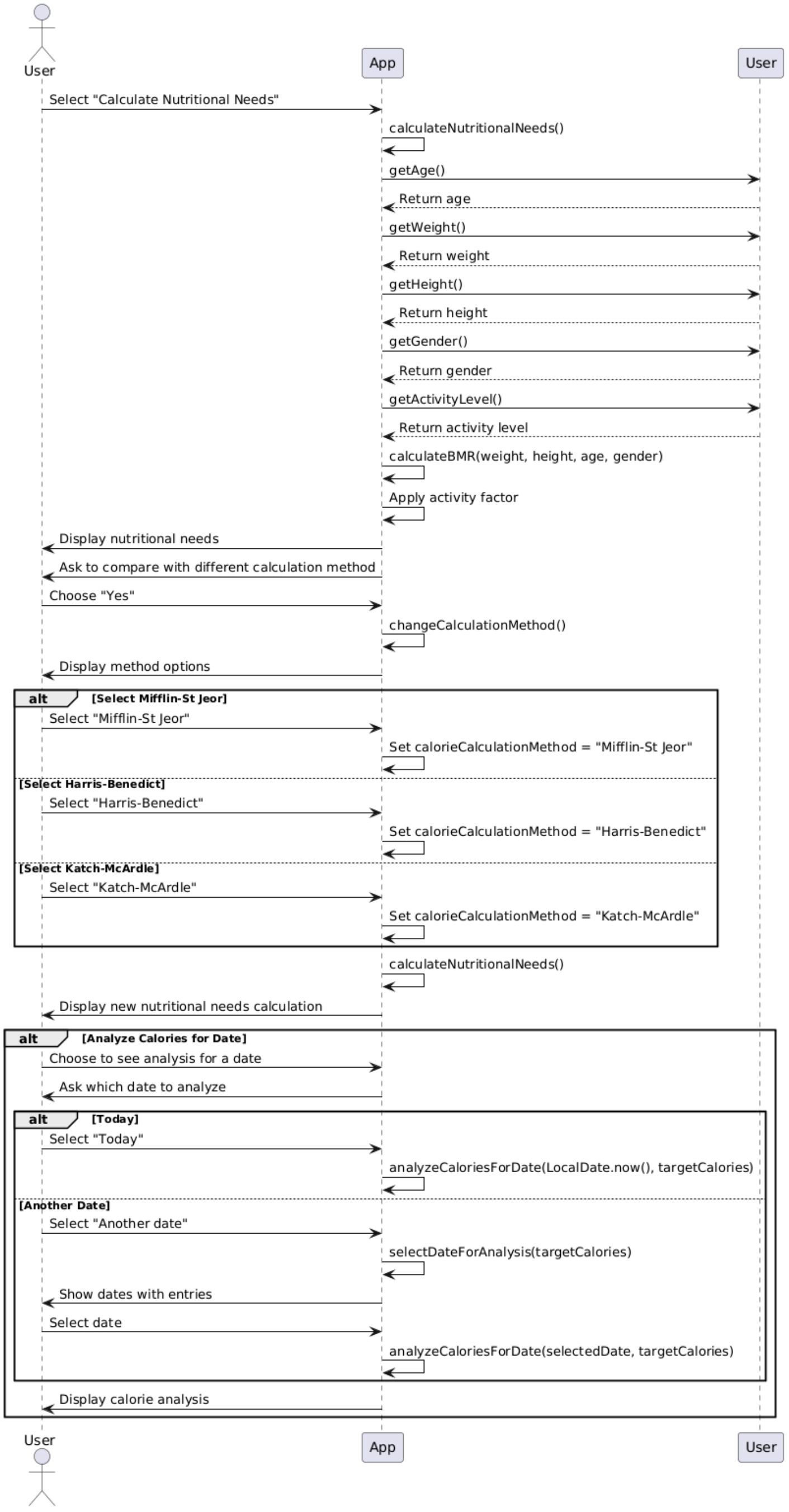
UPDATE EXISTING FOOD LOG ENTRY



PROFILE FLOW


User
App
User
Select "My Profile"
viewMyProfile()
getUserName()
Return username
getGender()
Return gender
getHeight()
Return height
getWeight()
Return weight
getAge()
Return age
getActivityLevel()
Return activity level
Calculate BMI
Display profile information
Show profile options
alt [Update Profile]
Select "Update Profile"
updateProfile()
Show update options
Choose field to update
alt [Update Weight]
Input new weight
setWeight(newWeight)
[Update Height]
Input new height
setHeight(newHeight)
[Update Age]
Input new age
setAge(newAge)
[Update Activity Level]
Select new activity level
setActivityLevel(newActivityLevel)
saveChangesToDatabase()
Confirmation
Confirm profile updated
[Change Calculation Method]
Select "Change Calculation Method"
changeCalculationMethod()
Display method options
Select calculation method
Set calorieCalculationMethod
Confirm method changed
[Return]
Select "Return to Home Menu"
User
App
User

CHANGE CALCULATION METHOD


User
App
User
Select "Calculate Nutritional Needs"
calculateNutritionalNeeds()
getAge()
Return age
getWeight()
Return weight
getHeight()
Return height
getGender()
Return gender
getActivityLevel()
Return activity level
calculateBMR(weight, height, age, gender)
Apply activity factor
Display nutritional needs
Ask to compare with different calculation method
Choose "Yes"
changeCalculationMethod()
Display method options
alt [Select Mifflin-St Jeor]
Select "Mifflin-St Jeor"
Set calorieCalculationMethod = "Mifflin-St Jeor"
[Select Harris-Benedict]
Select "Harris-Benedict"
Set calorieCalculationMethod = "Harris-Benedict"
[Select Katch-McArdle]
Select "Katch-McArdle"
Set calorieCalculationMethod = "Katch-McArdle"
calculateNutritionalNeeds()
Display new nutritional needs calculation
alt [Analyze Calories for Date]
Choose to see analysis for a date
Ask which date to analyze
alt [Today]
Select "Today"
analyzeCaloriesForDate(LocalDate.now(), targetCalories)
[Another Date]
Select "Another date"
selectDateForAnalysis(targetCalories)
Show dates with entries
Select date
analyzeCaloriesForDate(selectedDate, targetCalories)
Display calorie analysis
User
App
User

Design Principles

The YADA application demonstrates several key software design principles:

1. Separation of Concerns

Classes have clear, focused responsibilities:

- `User` handles user-related data and authentication
- `Food` and subclasses focus on food representation
- `DietLog` manages the logging of food consumption
- `FoodDatabase` manages food storage and retrieval
- `FoodImporter` interface and implementations handle external data sources

2. Low Coupling and High Cohesion

- Classes interact through well-defined interfaces
- The `Food` hierarchy uses abstraction to minimize dependencies
- Components like `FoodImporter` can be extended without modifying existing code
- The `DietLog` system operates independently from food creation

3. Information Hiding

- Implementation details are hidden behind interfaces
- Web food import details are encapsulated within importer implementations
- The `CompositeFood` class hides the complexity of managing ingredient components

4. Open-Closed Principle

- The design is open for extension but closed for modification
- New food importers can be added without changing existing code
- Food types can be extended with minimal impact on other components

5. Law of Demeter

- Objects communicate primarily with their immediate collaborators
- The `App` class delegates operations to appropriate managers rather than manipulating nested objects
- Components request services rather than querying for state and making decisions

6. Design Patterns

- **Composite Pattern**: `BasicFood` and `CompositeFood` sharing a common `Food` interface
- **Adapter Pattern**: `FoodImporter` implementations adapting external data sources
- **Command Pattern**: `UndoAction` for the undo feature
- **Factory Method**: Static creation methods in food classes for JSON parsing

Design Strengths and Weaknesses

Strongest Aspects

1. **Extensible Food Import System** The adapter-based design for food importers allows the system to easily incorporate new external data sources without modifying existing code. Each importer encapsulates the specific details of an external API, and the system can be extended simply by implementing the `FoodImporter` interface.
2. **Composite Food Structure** The food class hierarchy, using the composite pattern, provides a unified way to work with both basic foods and complex recipes. This allows for recursive composition of foods (recipes within recipes) while maintaining a consistent interface for calorie calculations and other operations.

Weakest Aspects

1. **User Interface Coupling** The UI logic in the `App` class is tightly coupled with business logic. A better approach would be to separate the presentation layer from business logic using an MVC pattern or similar architecture, enabling better testability and potential for alternative interfaces.
2. **Limited Data Validation** The current implementation has minimal validation for user inputs and food data. A more robust implementation would include comprehensive validation to ensure data integrity, such as checking for negative calorie values, valid height/weight ranges, and proper date formats.

KEY DESIGN POINTS

1. whenever user is adding a new food, he would be choosing from the given two options:
    a. directly add the calorie value
    b. provide the nutritional info of the food
        i. calories would be calculated from the nutritional info provided.

    Based on the option chosen by the user, different constructors for the FOOD class and its sub classes would be invoked. ADD new factory methods for json parsing including the appropriate fields for the nutritional info
2. For calculating the target calories, if we want to add a new method just we have to add an extra case in the switch case in the method choosing function and extra case in the switch case that chooses the formula for the calculation.
3. we have separated the data layer from the business logic layer, so the data layer fetches the data from any source and it should provide a mapping function that maps any data format to the format mentioned in the food class of the main business layer. so, to add any new data source the developer should just add the new class that contains :
    a. data fetching function(to fetch the raw data)
    b. mapping function(which maps raw data to the data format supported in the business logic layer).
4. To efficiently store the info in the log file, when adding an item to a particular day’s log, the log will be searched to find if the same item already exists, if it exists the servings of the existing item is updated to the total servings by adding the new servings to existing servings, instead of creating a new entry.