YADA Design Document

Project Team

- **Team Members:** Akmal Ali, Rohan
- **Date:** April 8, 2025

Product Overview

YADA is a command-line interface (CLI) diet management application that helps users track their food intake, calculate caloric needs, and manage nutritional goals. The application allows users to:

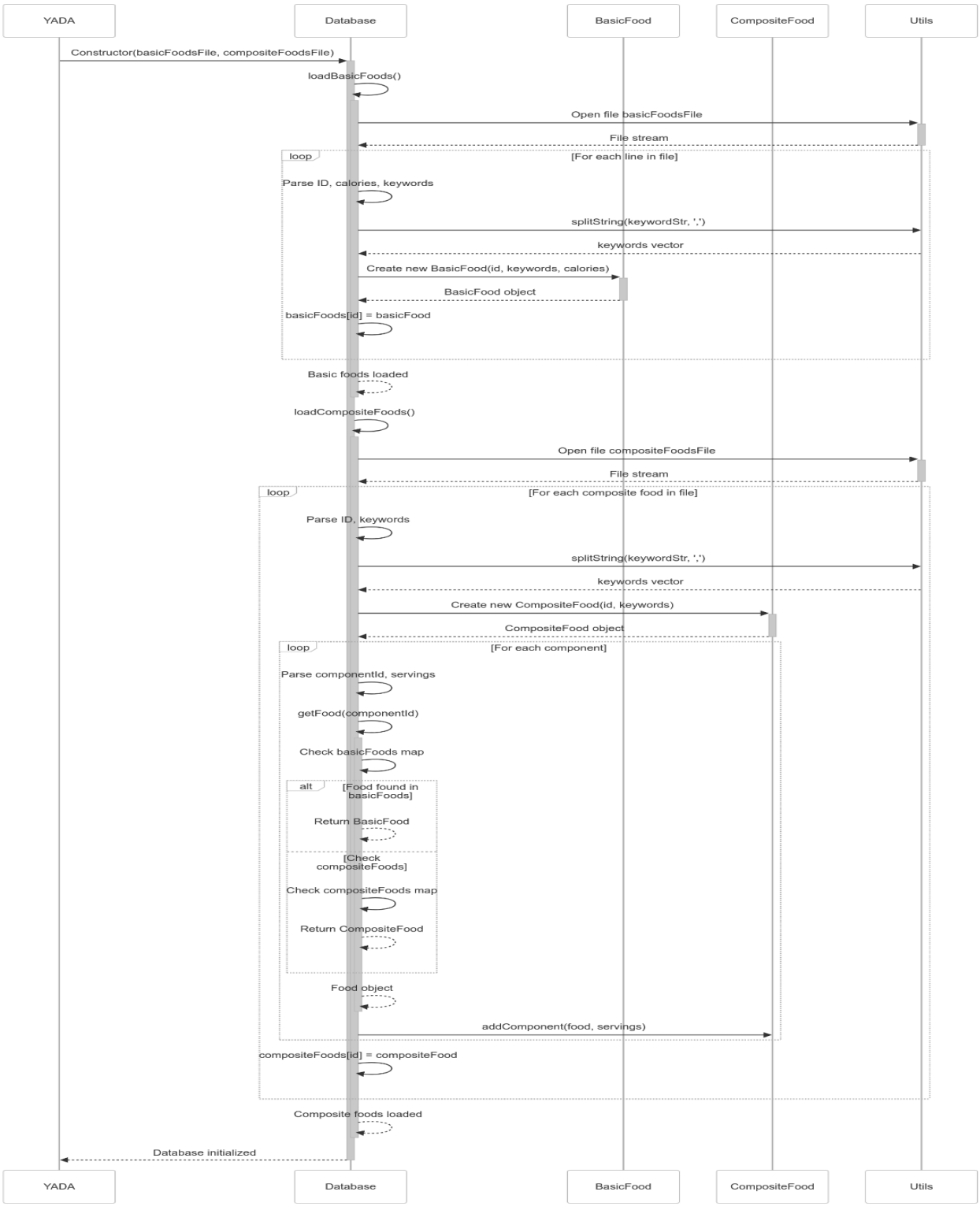
- Create and manage user profiles with personalized information
- Track daily food consumption
- Calculate BMR (Basal Metabolic Rate) and TDEE (Total Daily Energy Expenditure)
- Search and add both basic and composite food items to a database
- View nutritional summaries and intake logs
- Maintain food consumption history with undo functionality

The application is designed with modularity and extensibility in mind, using object-oriented design principles to create a maintainable and scalable architecture.

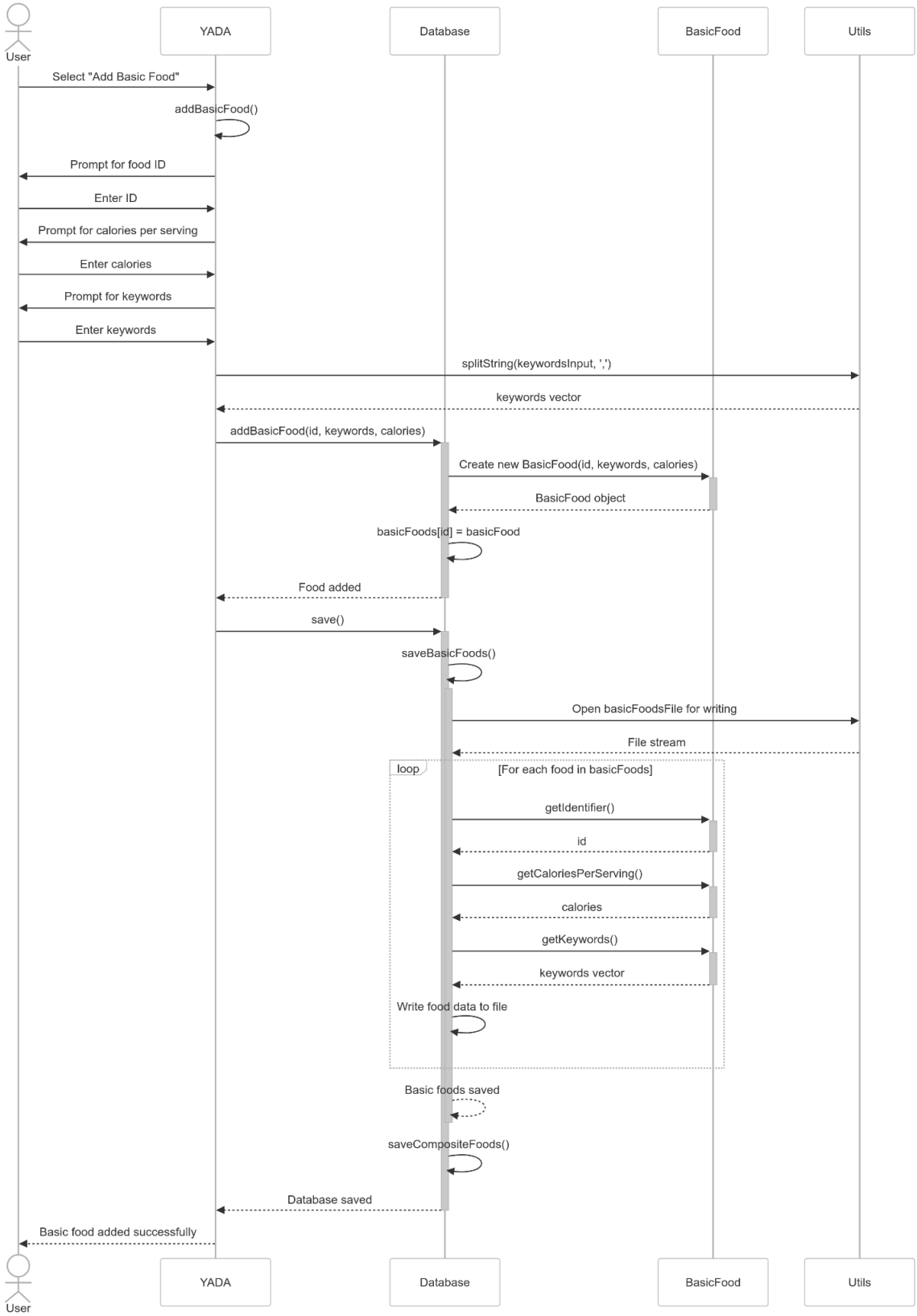
UML Class Diagram

Sequence Diagrams

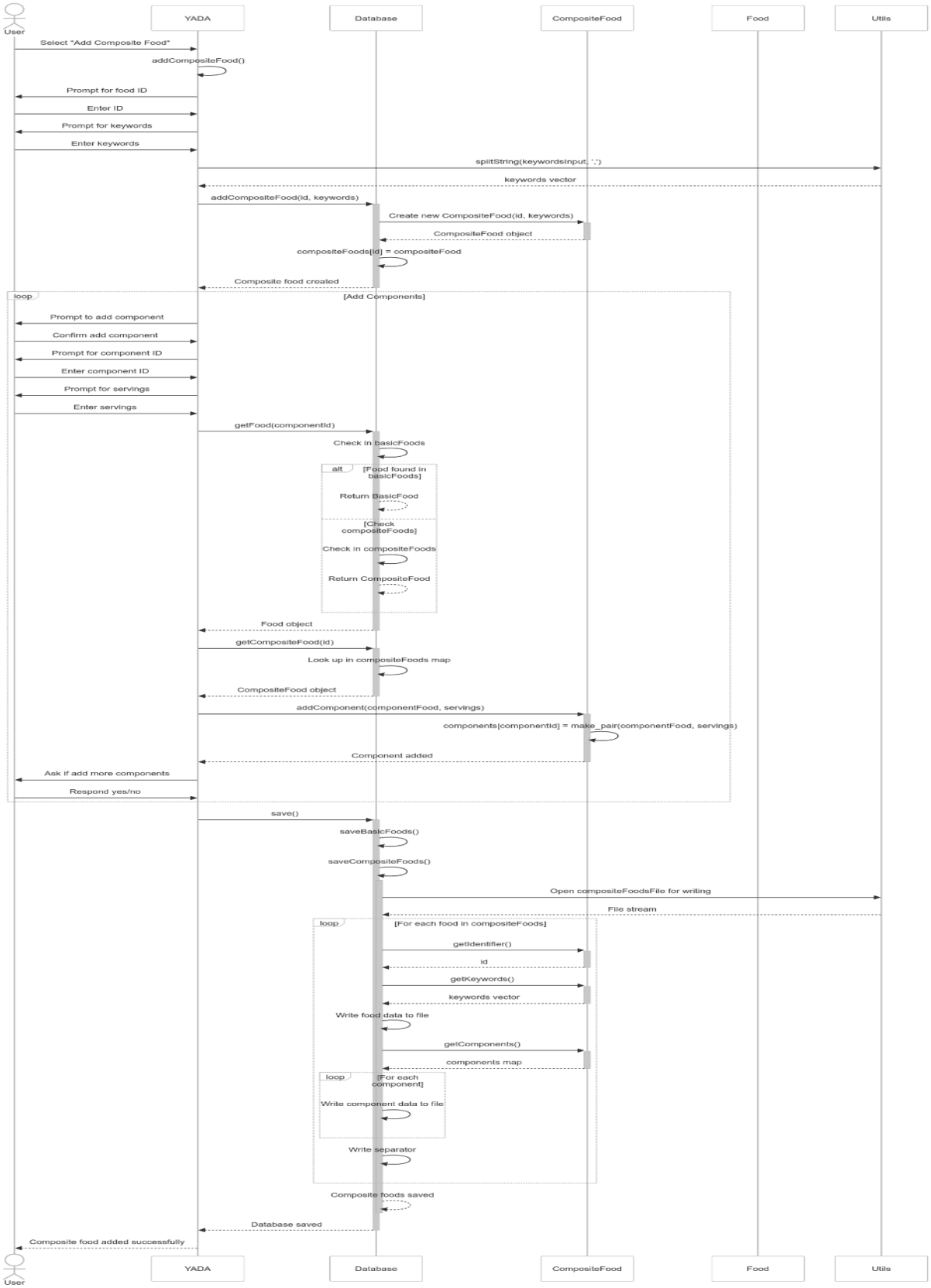
1: Database Initialization and Loading Foods



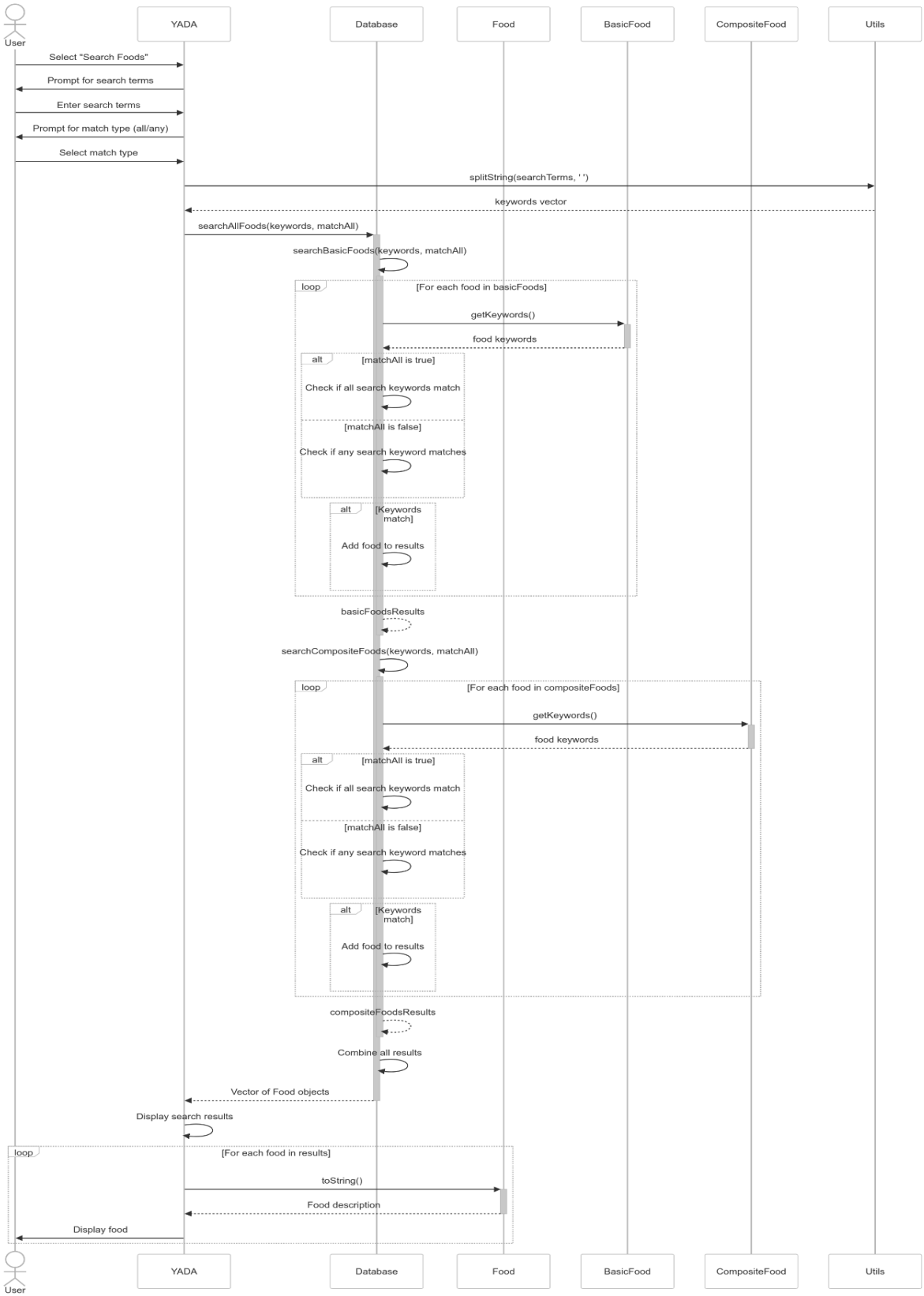
2: Adding a Basic Food to Database



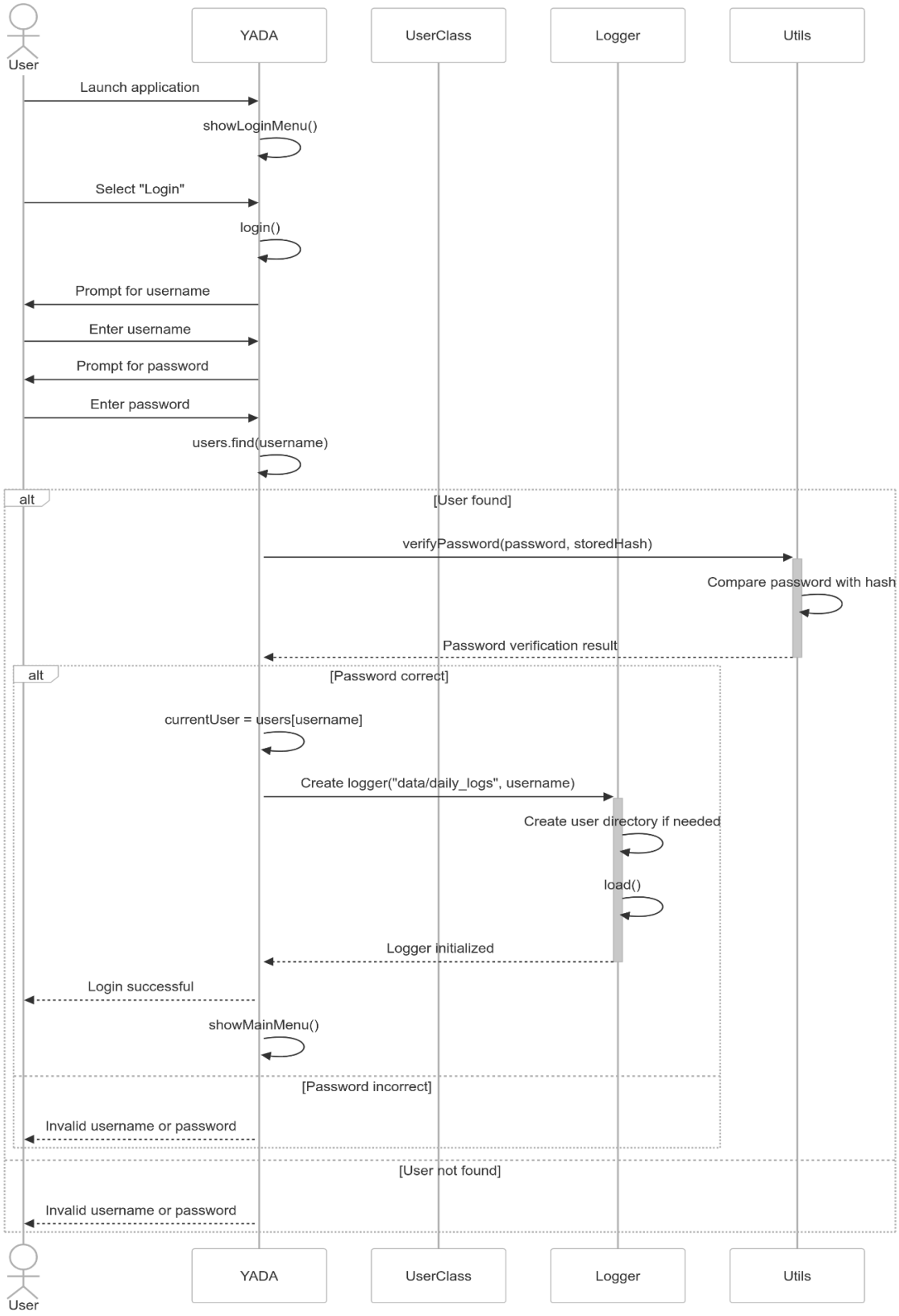
3: Adding a Composite Food to Database



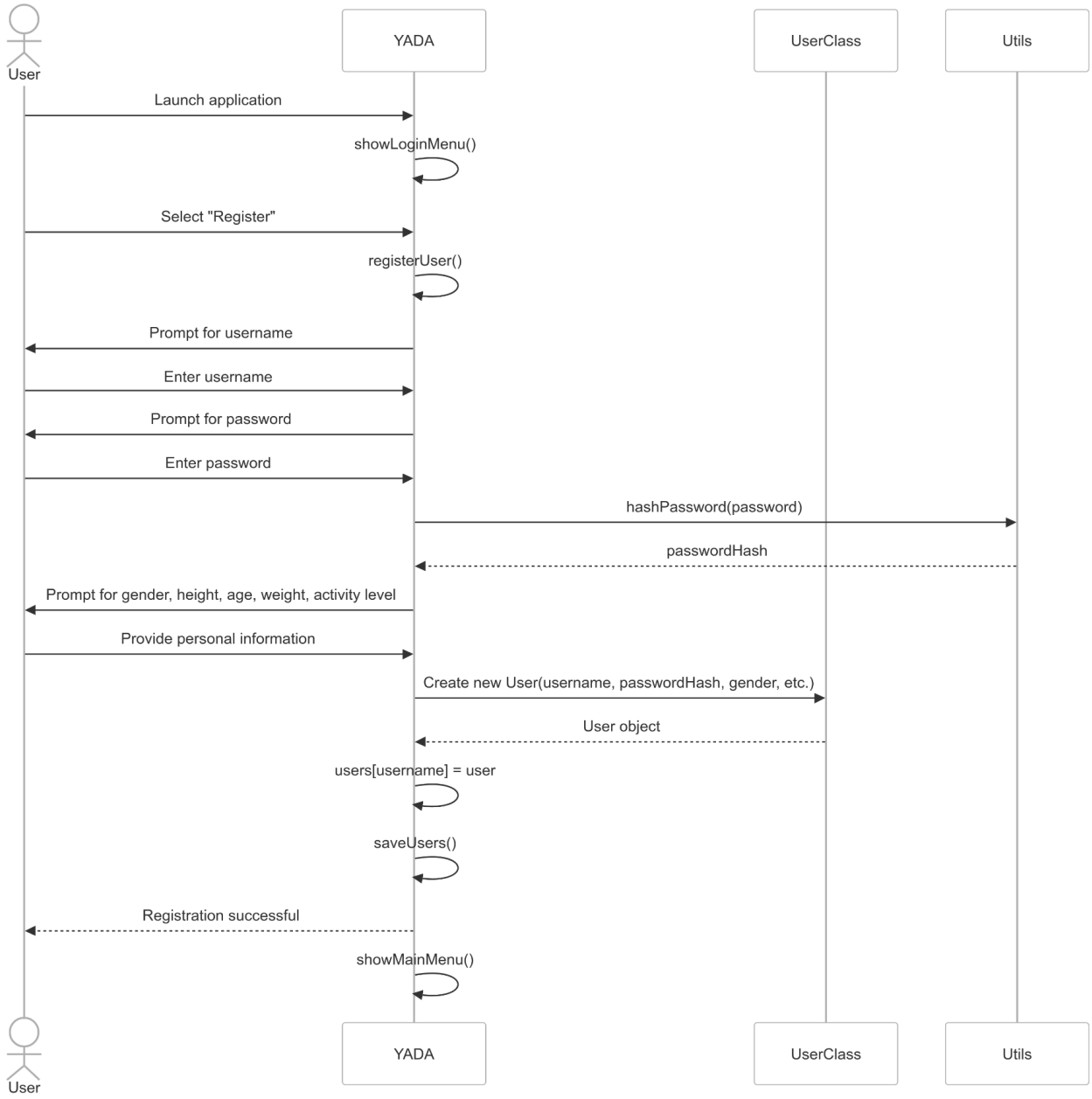
4: Searching Foods in Database



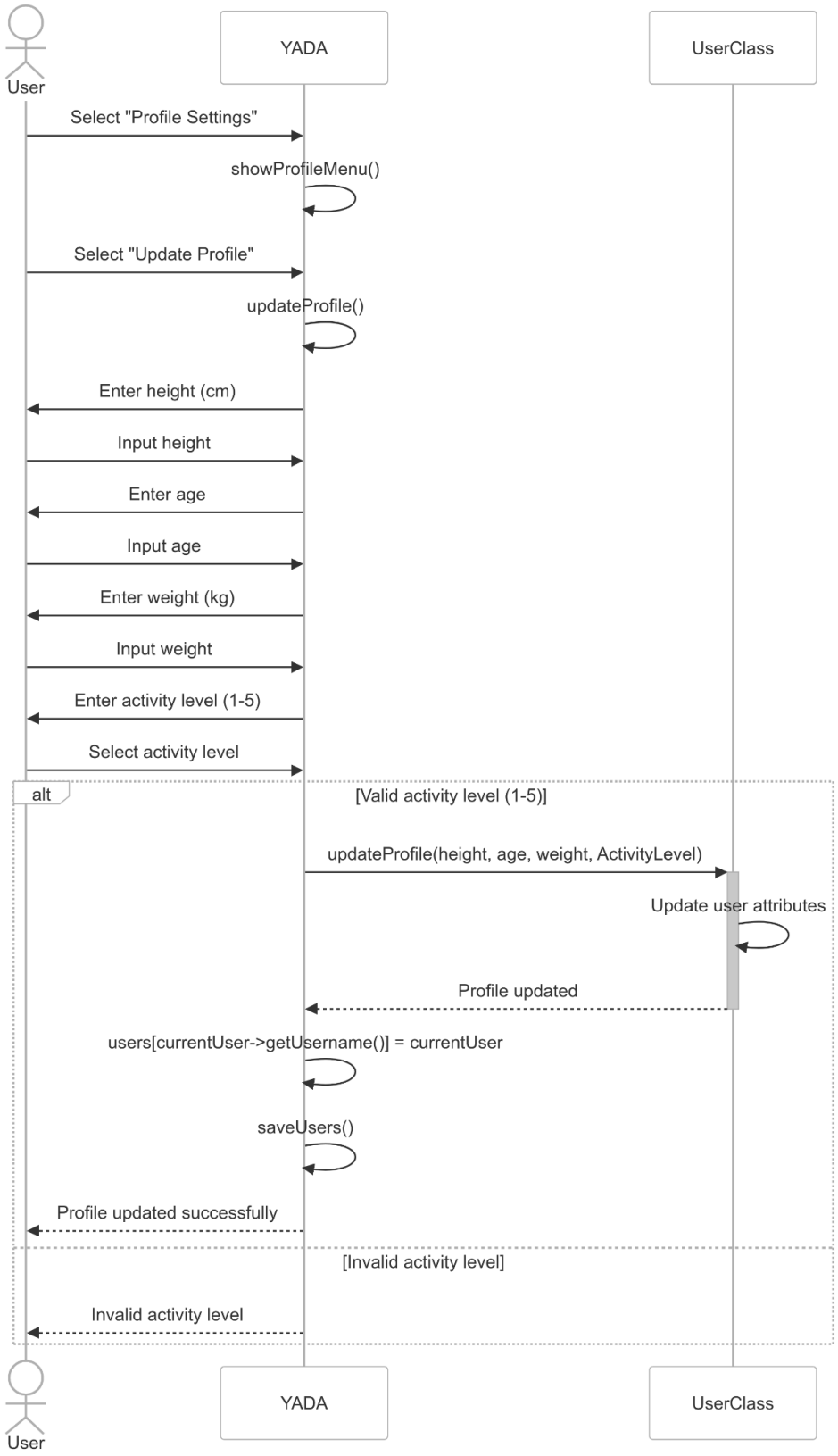
5: User Login Flow



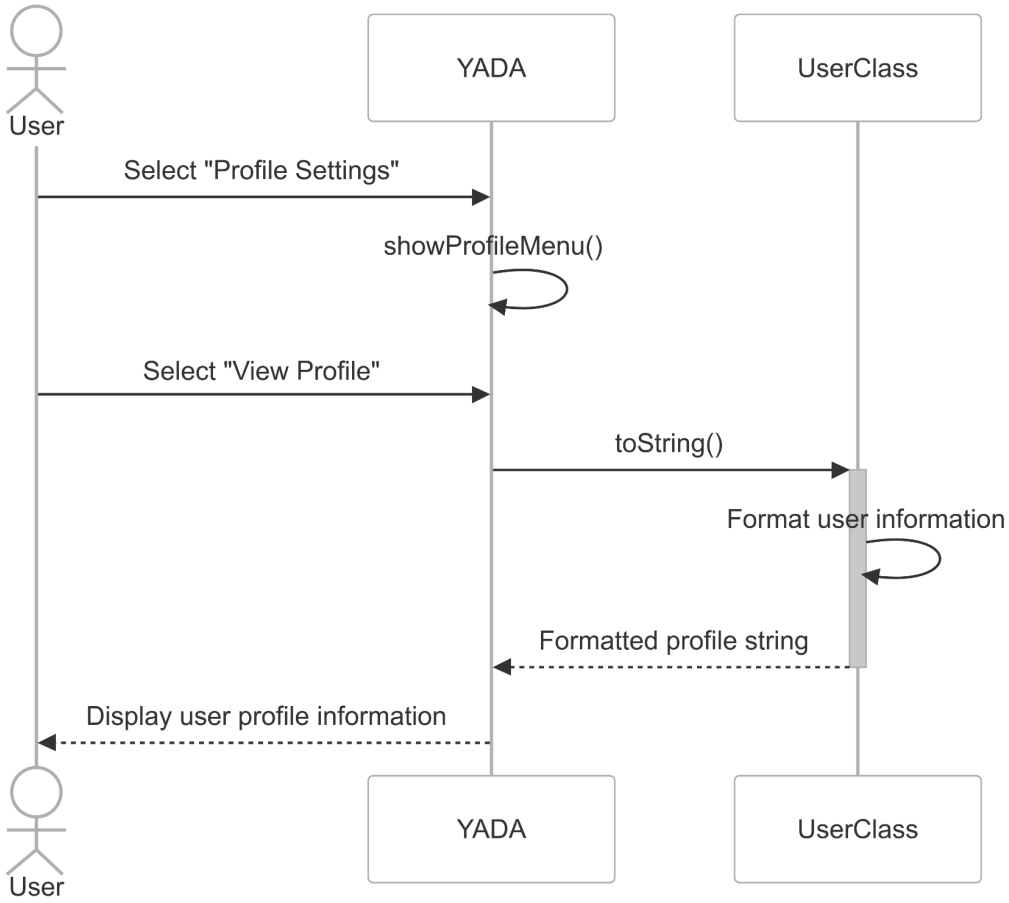
6: User Registration Flow



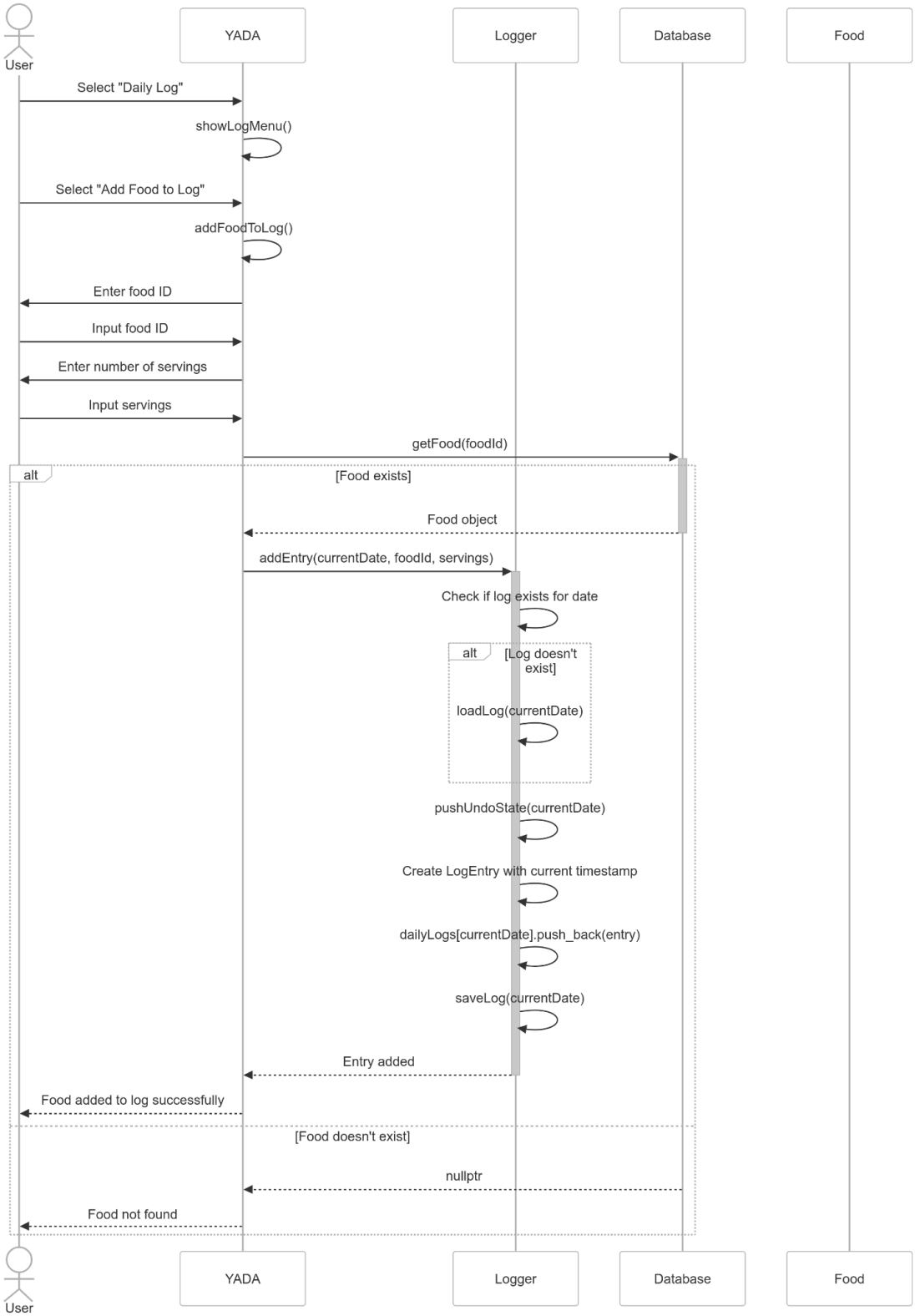
7: Update User Profile



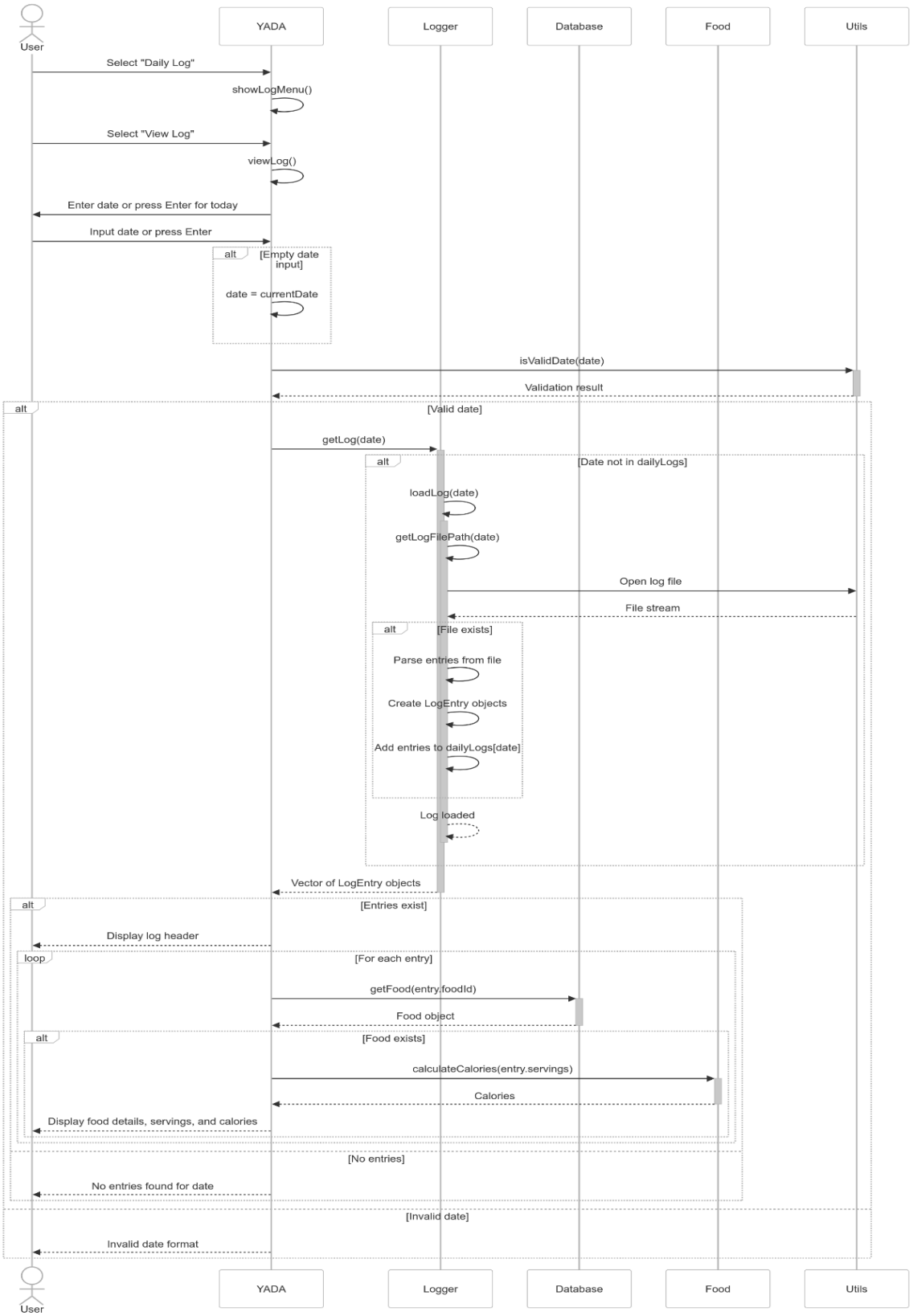
8: View Profile



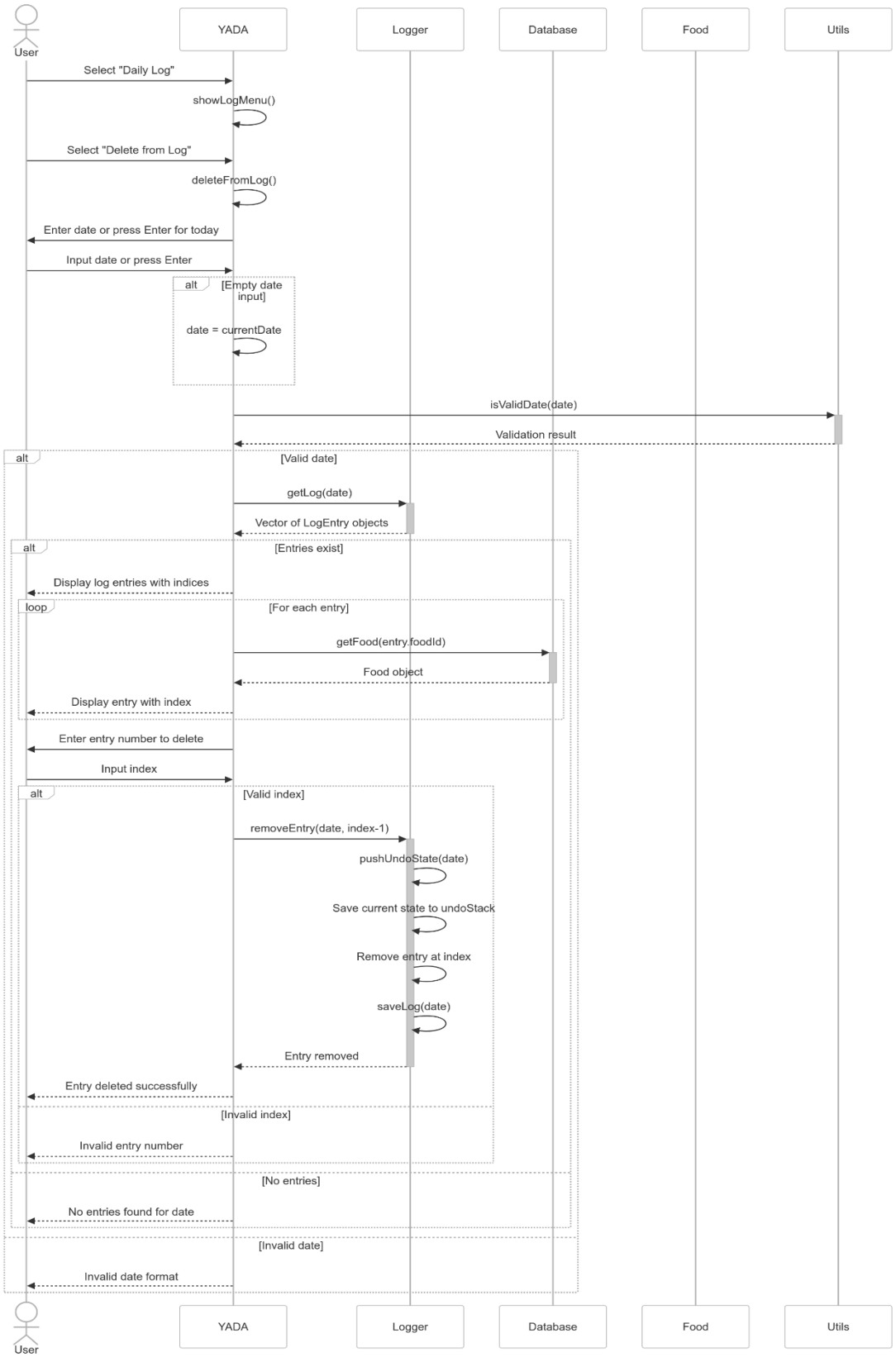
9: Add Food to Log



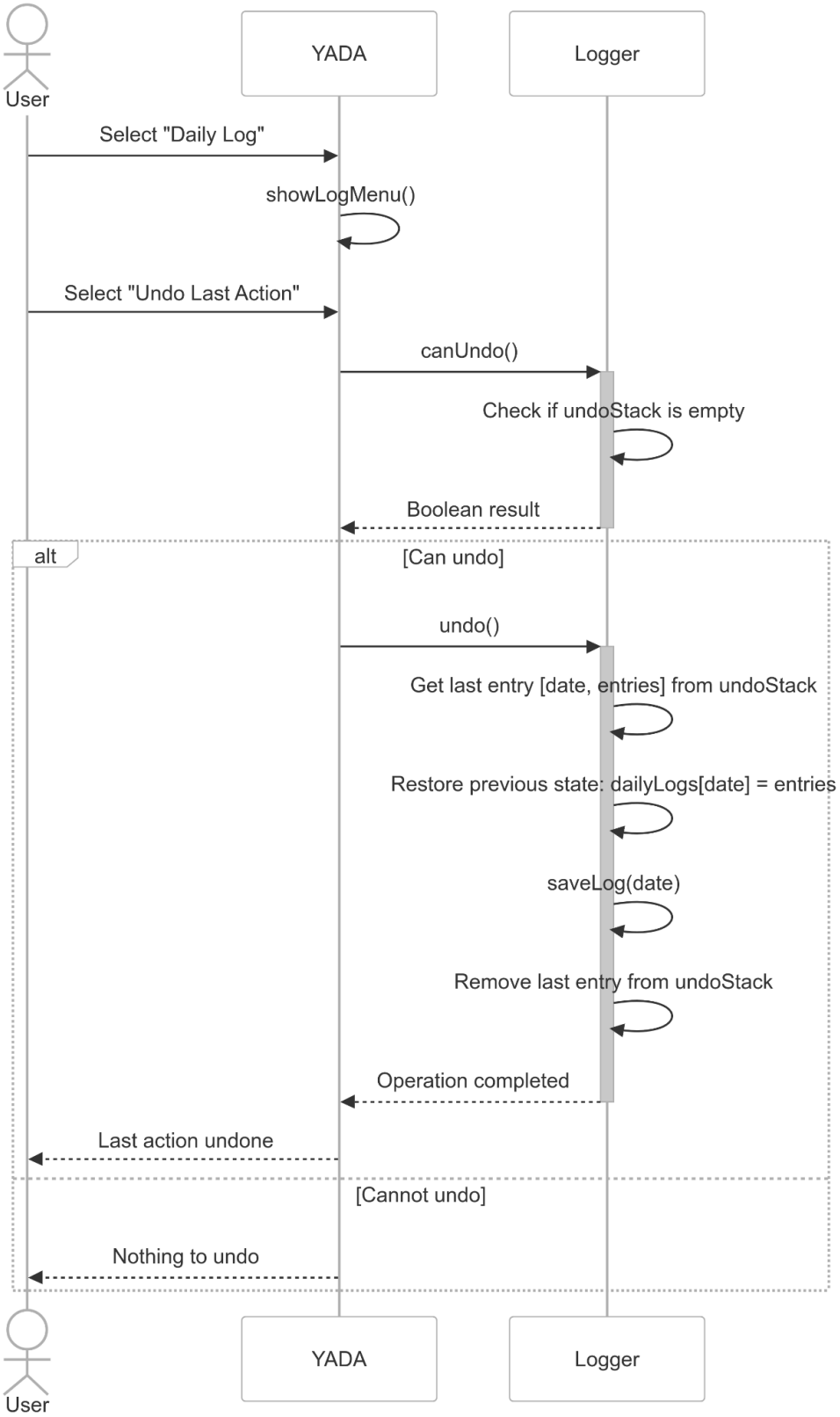
10: View Log


User
YADA
Logger
Database
Food
Utils
Select "Daily Log"
showLogMenu()
Select "View Log"
viewLog()
Enter date or press Enter for today
Input date or press Enter
alt
[Empty date input]
date = currentDate
isValidDate(date)
Validation result
alt
[Valid date]
getLog(date)
alt
[Date not in dailyLogs]
loadLog(date)
getLogFilePath(date)
Open log file
File stream
alt
[File exists]
Parse entries from file
Create LogEntry objects
Add entries to dailyLogs[date]
Log loaded
Vector of LogEntry objects
alt
[Entries exist]
Display log header
loop
[For each entry]
getFood(entry.foodId)
Food object
alt
[Food exists]
calculateCalories(entry.servings)
Calories
Display food details, servings, and calories
[No entries]
No entries found for date
[Invalid date]
Invalid date format

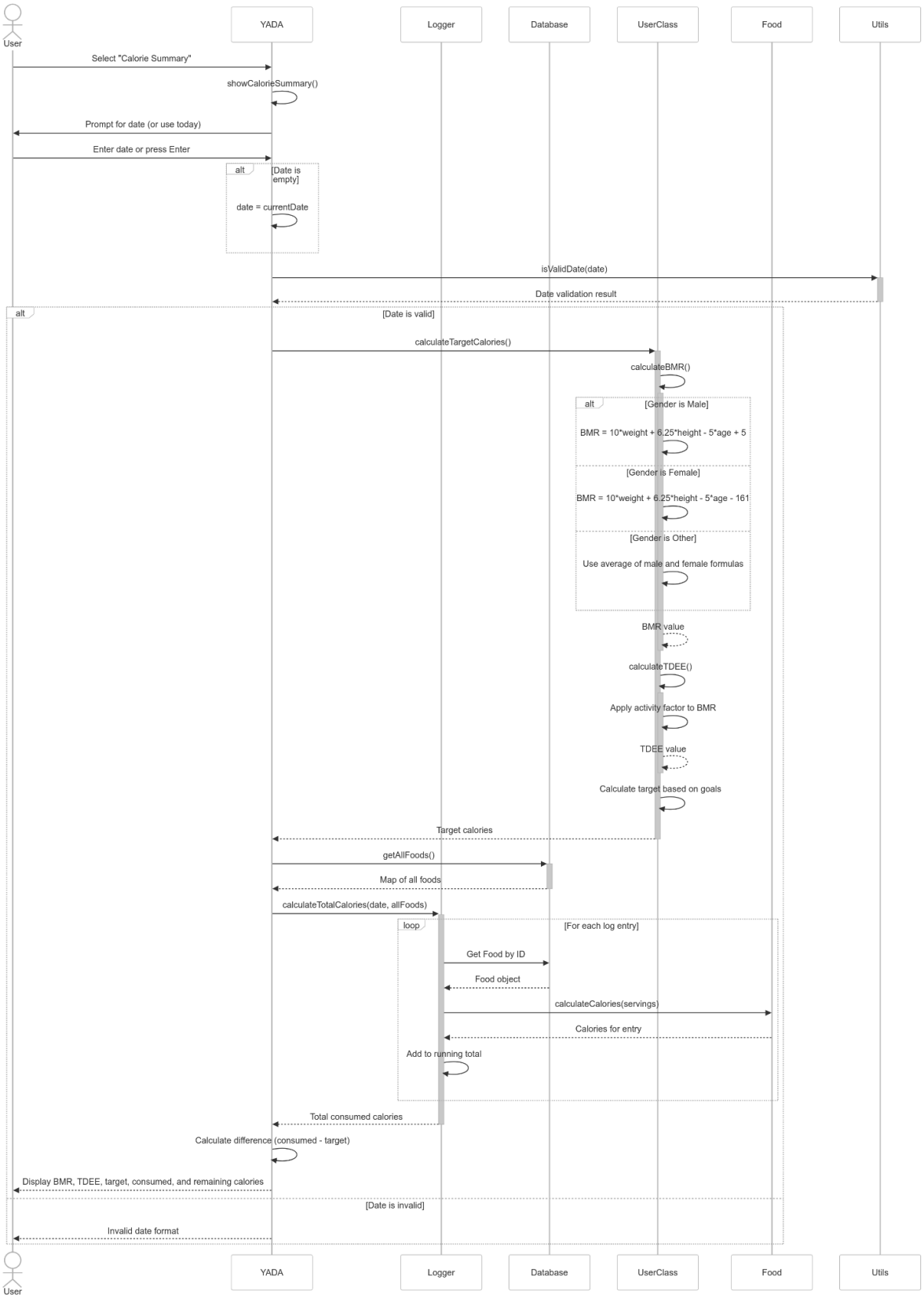
11: Delete from Log



12: Undo Last Action



13: Show Calorie Summary



Design Principles Analysis

Low Coupling

The YADA application demonstrates low coupling through its well-defined component boundaries:

- **User Component**: Manages user profile data and calorie calculations (BMR, TDEE) without direct dependencies on other system components
- **Food Hierarchy**: Implements the Composite pattern which allows treating `BasicFood` and `CompositeFood` uniformly without exposing implementation details
- **Logger Component**: Handles food log operations using only `foodId` references rather than direct food object dependencies
- **Database Component**: Manages food storage and retrieval with clearly defined interfaces

The system achieves low coupling through:

1. **Interface-Based Communication** – Components interact through well-defined interfaces rather than directly accessing implementation details
2. **Dependency Injection** – The YADA main class injects dependencies (like Logger and Database) into relevant operations
3. **Reference by ID** – Using string identifiers (`foodId`) to reference objects instead of direct object references
4. **Separate Persistence** – Each component manages its own persistence (`users.txt`, food files, and log files)

High Cohesion

Each class in the system has a single, well-defined responsibility:

- **User** – Manages user profile and calorie calculations (BMR/TDEE based on physical attributes)
- **Food** – Represents food items and their caloric values

- **Logger** – Handles daily food logs and undo functionality
- **Database** – Maintains the food database with search capabilities
- **Utils** – Provides utility functions that support other components

This high cohesion makes the system more maintainable since each component has a clear, focused purpose.

Separation of Concerns

The application effectively separates different concerns:

1. **Data Management** – `Database` and `Logger` components handle persistent storage
2. **Business Logic**
    - `User` component calculates BMR, TDEE, and target calories
    - Food hierarchy handles calorie calculations for different food types
3. **User Interface** – `YADA` class handles all user interactions and menu displays
4. **Authentication** – Login/registration components manage user credentials

Information Hiding

The design employs multiple techniques to hide implementation details:

1. **Private Member Variables** – All classes use private fields to restrict direct access
2. **Encapsulation** – Getters and setters control access to attributes
3. **Private Helper Methods** – Methods like `loadLog()` and `pushUndoState()` in `Logger` are hidden
4. **Abstraction** – The `Food` base class hides implementation differences between food types

Law of Demeter

The application generally respects the Law of Demeter (principle of least knowledge):

1. The `YADA` main class coordinates interactions between components rather than having them interact directly
2. Methods typically interact only with:
    - Their parameters
    - Objects they create
    - Their own instance variables
3. Components don't access other components through returned objects

This principle helps keep the system modular and reduces dependencies between components.

Design Patterns

Composite Pattern

Implemented in the food hierarchy where:

- `Food` is the abstract component
- `BasicFood` is the leaf component
- `CompositeFood` is the composite that contains other components

This allows treating individual foods and composite foods uniformly, supporting operations like calorie calculation regardless of food complexity.

Command Pattern

Implemented in the `Logger`'s undo functionality:

- The `undoStack` stores previous states

- `pushUndoState()` saves the current state before modifications
- `undo()` restores the previous state

This pattern allows the system to track and reverse operations.

Singleton Pattern (Implicit)

The `YADA` main application class functions as a de facto singleton, coordinating system functionality through a centralized interface, though it does not strictly enforce the Singleton pattern.

Strategy Pattern (Implied)

The calorie calculation method in `User` class suggests a Strategy pattern:

- Currently uses the Mifflin-St Jeor Equation
- Could be extended to support different calculation methods based on user preference

Design Strengths and Weaknesses

Strengths

1. **Extensibility**
    - New food types can be added by extending the `Food` base class
    - The `Logger`'s design supports adding new tracking features
    - The `User` class is designed to accommodate different calorie calculation methods
2. **Maintainability**
    - Clear separation of concerns makes the code easier to understand and modify
    - Each component has well-defined responsibilities
    - Consistent naming conventions and organization enhance readability

Weaknesses

1. **Limited Diet Goal Flexibility**
    - The system uses a fixed calculation for target calories (`TDEE - 500`) without support for different goals
    - There's no support for macronutrient tracking (protein, carbs, fat)
2. **Centralized Control**
    - The `YADA` main class handles too many responsibilities, potentially becoming a bottleneck
    - More delegation to specialized controller classes could improve the design

Reflection

Strongest Design Aspects

1. **Composite Food Model** – The implementation of the Composite pattern for food items is elegant and powerful, allowing unlimited nesting of composite foods while maintaining a consistent interface. This enables complex recipes to be built from basic ingredients with proper calorie calculation.
2. **Undo Functionality** – The `Logger`'s undo system provides a robust way to correct mistakes, enhancing user experience. The implementation stores previous states efficiently and supports multiple levels of undo.

Weakest Design Aspects

1. **Limited Nutritional Tracking** – The system currently only tracks calories without support for other nutritional information like macronutrients, vitamins, or minerals. This limits its usefulness for comprehensive diet management.
2. **Monolithic Controller** – The `YADA` class handles too many diverse responsibilities from UI to business logic coordination. This could be improved by implementing a more refined MVC architecture with separate controllers for different functional areas.

Conclusion

The YADA application demonstrates a thoughtful object-oriented design that balances different quality attributes. Its strengths in extensibility and maintainability make it a solid foundation for a diet management application. The use of design patterns like Composite and Command enhance its flexibility while maintaining code organization.

Future improvements could focus on addressing the identified weaknesses by adding support for more comprehensive nutritional tracking and refining the architecture to better distribute responsibilities among components.