Milestone 2: Project design using UML diagrams

By Ryan Davis, Ivan Farfan, Johan Jaramillo, and Cory Vitanza

*We pledge our Honor that we have abided by the Stevens Honor System*

Table of Contents

[Introduction 2](#_Toc192541783)

[Team Collaboration 3](#_Toc192541784)

[Use Case Diagram 4](#_Toc192541785)

[Activity Diagram 7](#_Toc192541786)

[Class Diagram 9](#_Toc192541787)

[Sequence Diagram 10](#_Toc192541788)

[Discussions of Lessons Learnt from this Milestone 11](#_Toc192541789)

# Introduction

Our group was tasked with creating a meal sharing application that allows users to host and join meals. Users can assume both roles under one account, and hosts are able to share information on a meal and invite others to join them while guests can view meals nearby and request to join the meal. The current milestone employs the following UML diagrams: use case diagram, activity diagram, class diagram, and sequence diagram. In doing so, the implementation of the design will be much smoother and more efficient, as the group will have an improved understanding of the system design.

# Team Collaboration

* Ryan Davis- Sequence Diagram
* Ivan Farfan- Class Diagram
* Johan Jaramillo- Use Case Diagram
* Cory Vitanza- Activity Diagram

# Use Case Diagram

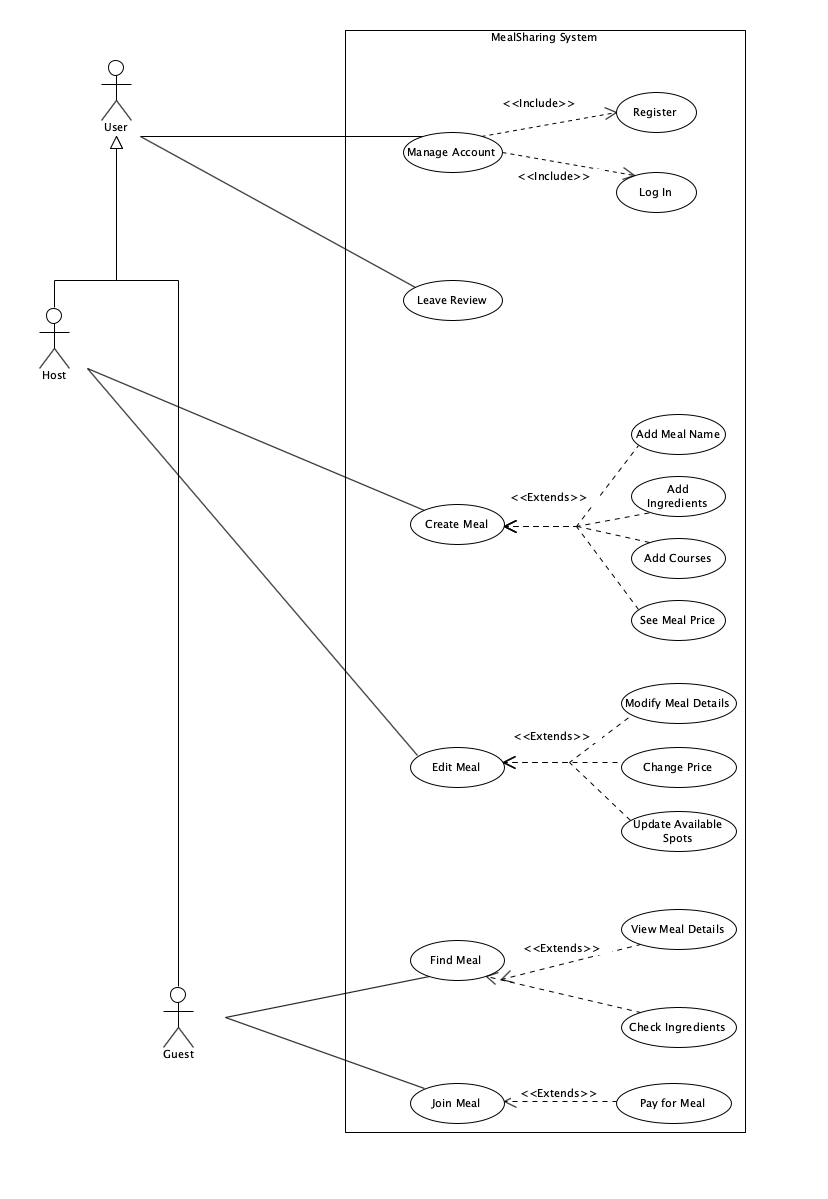


Figure 1: Use case Diagram

Use Case Description for Meal Sharing Application

Actors:

1. User: A general actor that manages accounts and leaves reviews.
2. Guest: A user who finds and joins meals.
3. Host: A user who creates and edits meals.

Use Cases:

**UC1: Manage Account**

Actor: User

Description: Allows a user to manage their account, including registering and logging in.

Includes:

* UC1.1: Register: Users can create an account by providing their name and email and then creating a password.
* UC1.2: Log In: Users can log into their existing accounts to access the platform.

**UC2: Leave Review**

Actor: User

Description: Both the host and the guest can leave a review for each other after completing a meal, rating their experience and sharing feedback about the food.

**UC3: Create Meal**

Actor: Host

Description: Hosts can create new meal listings, defining details about the meal they are offering.

Includes:

* UC3.1: Add Meal Name: Hosts specify the name of the meal.
* UC3.2: Add Ingredients: Hosts list the ingredients used in the meal.
* UC3.3: Add Courses: Hosts define whether the meal includes multiple courses.
* UC3.4: Set Meal Price: Hosts can specify the price they will charge for joining their meal.

**UC4: Edit Meal**

Actor: Host

Description: Hosts can modify existing meal listings to update details.

Extends:

* UC4.1: Modify Meal Details – Hosts can update the meal description, time, and location.
* UC4.2: Change Price – Hosts can adjust the cost of the meal if necessary.
* UC4.3: Update Available Spots – Hosts can modify the number of available spots for guests.

**UC5: Find Meal**

Actor: Guest

Description: Guests can search for available meals based on their preferences.

Extends:

* UC5.1: View Meal Details: Guests can see meal descriptions, host information, and availability.
* UC5.2: Check Ingredients: Guests can review ingredient lists for dietary restrictions or allergies.

**UC6: Join Meal**

Actor(s): Guest

Description: Guests can join a meal listed by a host.

Extends:

* UC6.1: Pay for Meal: Guests can complete payment before joining.

# Activity Diagram

A diagram of a restaurant menu

AI-generated content may be incorrect.

This activity diagram outlines the user journey in the meal-sharing app, covering login, meal hosting/joining, payments, and rating interactions.

**1. Start & Login Flow**

**The process starts** at the Login Page. A decision node checks if the user is already logged in:

* Yes: User goes directly to the Dashboard/Homepage.
* No: User is redirected to the Signup/Login screen.

**2. Choosing an Action: Hosting or Joining a Meal**

From the Dashboard/Homepage, the user decides whether to:

* Host a meal
  + The user creates a meal listing.
  + Adds meal details (e.g., name, time, location).
* Join a meal
  + The user searches for available meals.
  + Views meal details and chooses to join.

**3. Handling Payments (If required)**

* A decision point checks if the meal requires payment.
* If payment is needed:
  + The user adds a payment method (if they haven’t already).
  + Select a payment method.
  + Confirms payment.
  + If payment fails, they must retry.
  + If successful, they proceed to the Confirmation Screen.
* If payment isn’t required, they go directly to the Confirmation Screen.

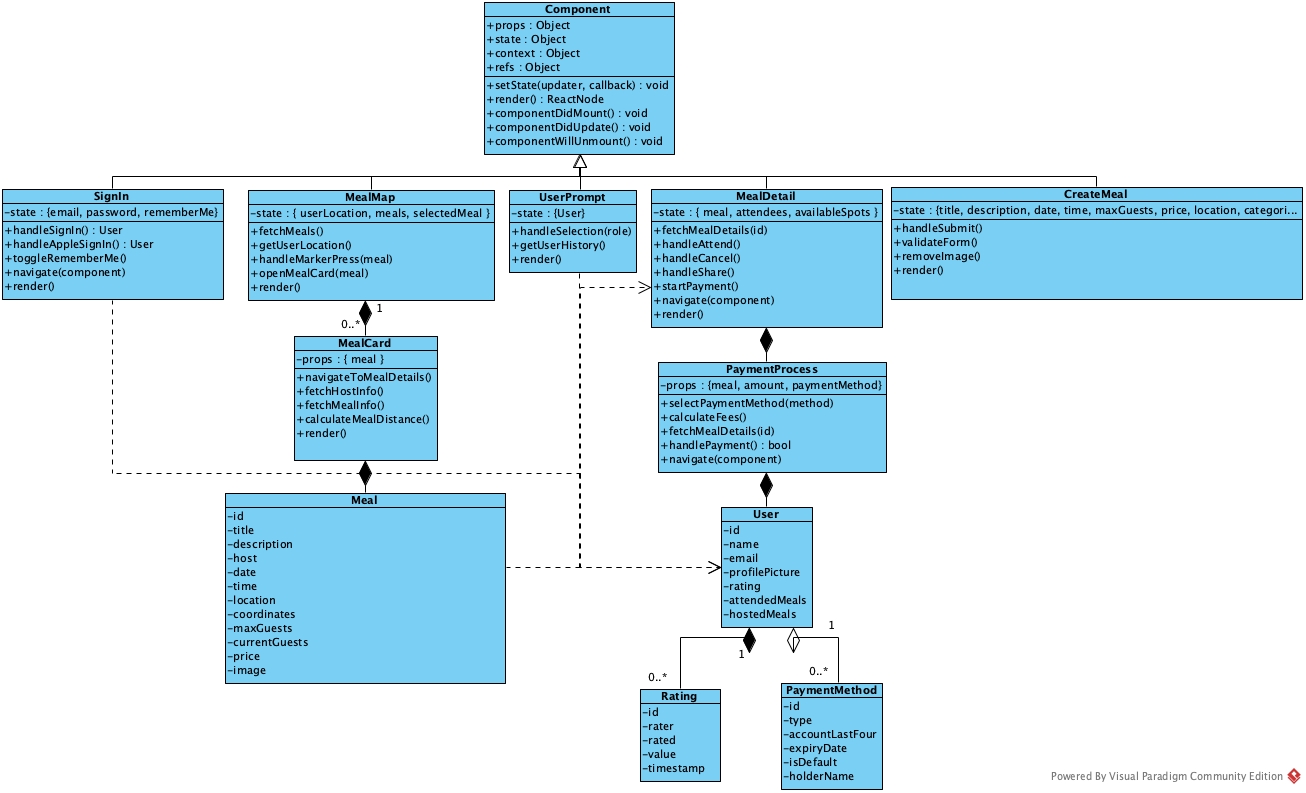
**4.** **Meal Completion and Ratings**

* After the meal, the system checks if the meal is completed.
* If yes, the system prompts ratings:
  + Guests rate the host/meal.
  + Hosts rate the guest.
* If ratings are skipped, the flow moves to End.

**5. End of Flow**

Once the user completes the meal and ratings, **the workflow ends.**

# Class Diagram

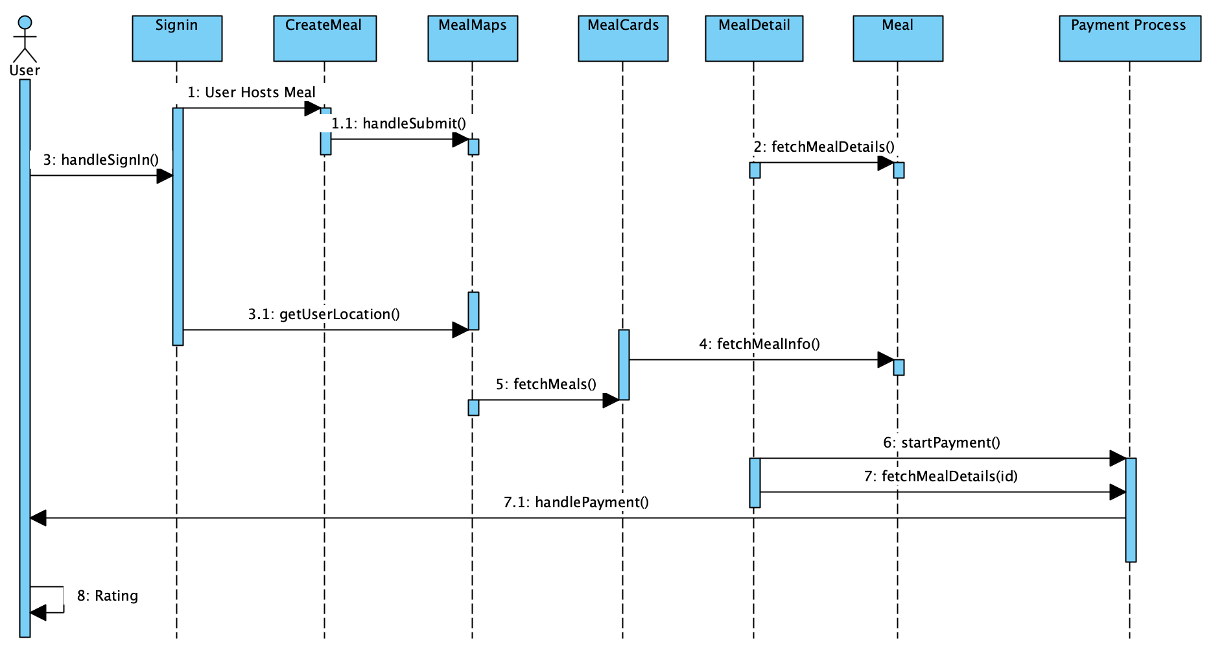


The diagram shown above illustrates the object (class) architecture our group came up with. Since one of our first technical decisions was to use the React Native framework to build our application, we began by adding the Class Component from React, that is later inherited to the diverse components of the application, providing the render and displaying related methods that each “screen” needs to be visible as well as the attributes to pass states. The list of class components defined include Sign In (SignIn), Home Screen – prompts user to select view of Host or Guest, View of Map – including Meal markers (MealMap), details about a specific meal (to register for it) (MealDetail) and the screen of meal creation (CreateMeal).

Using these components as a basic unit of our class diagram, we explored the relationships between the class components and the other classes, such is the case of the class User, that defines the structure of each User, and we indicate that SignIn, UserPrompt and MealDetail have a dependency relationship on User, that means that there’s no ownership of classes between them but rather that the class components need user for their operations. This is the case of the dependency of MealDetail on Meal, where to display the information of a meal, the class component MealDetail needs to fetch the data from a Meal object.

We also considered composition relationships, where we indicate that a single MealMap can have zero to many MealCard(s), also, each MealCard is composed of a meal. Similarly, each PaymentProcess is part of a MealDetail, this means that every time a user accesses the details of a meal, a new payment process is created awaiting further input. A user is also tightly bound to the PaymentProcess, as it follows a composition relationship where a payment process must always be associated with a specific user. Finally, each user is composed of one or multiple ratings that are later computed to display a numerical rating. Each user can have zero or multiple payment methods, but its aggregation relationship between the two indicates how the payment method existence does not depend on the user class.

# Sequence Diagram



The sequence diagram above provides the process flows of the meal sharing application. On launch, the user is prompted to sign in. After signing in, the user has the option to create or host a meal, or join a meal, where the user’s location is received. After the meal is created, the user can view meals, and fetch meals from the meal cards. The meal cards receive the details of the meal from the meal class. After joining a meal and viewing the meal’s details, the user can begin their payment based on the meal's details. The user then receives Boolean verification that the payment was completed, allowing for confirmation of their reservation.

# Discussions of Lessons Learnt from this Milestone

From this milestone, the group can design the system to better understand the system design. System design guarantees that software development begins with a comprehensive plan, minimizing future mistakes and ensuring ease of maintenance. Documentation is essential in an iterative and team-based design process since properly structured UML diagrams enhance communication among team members as well as serve as a reference point for future changes. To ensure accuracy and readability, diagram notation consistency is required because it may lead to confusion and design flaws. Secondly, modularity among different UML diagrams during software design allows each of the diagrams to focus on a single aspect of the system, thereby making the design, scalability, and maintenance with evolving requirements easier.