System Design Document v1.0

Product: Waste Watcher

Customer: Senator George J. Mitchell Center for Sustainability Solutions

Development Team: Kayak Development Solutions

Team Members:

* Declan Brinn
* Gavin Palazzo
* Levi Sturtevant
* Chase Pisone
* Finn Jacobs

**Table of Contents**

[1 Introduction](#_heading=h.gjdgxs) 3

[1.1 Purpose of This Document](#_heading=h.30j0zll) 3

[1.2 References](#_heading=h.1fob9te) 3

[2 System Architecture](#_heading=h.3znysh7) 3

[2.1 Architectural Design](#_heading=h.2et92p0) 4

[2.2 Decomposition Description](#_heading=h.tyjcwt) 5

[3 Persistent Data Design](#_heading=h.3dy6vkm) 9

[3.1 Database Descriptions (if you use a database)](#_heading=h.1t3h5sf) 9

[3.2 File Descriptions](#_heading=h.4d34og8) 11

[4 Requirements Matrix](#_heading=h.2s8eyo1) 11

[Appendix A – Agreement Between Customer and Contractor](#_heading=h.17dp8vu) 12

[Appendix B – Team Review Sign-off](#_heading=h.3rdcrjn) 13

[Appendix C – Document Contributions](#_heading=h.26in1rg) 14

# **Introduction**

This is a capstone project for Susanne Lee, Travis Blackmer, and Ryan Fitzmaurice in partial fulfillment of the Computer Science BS degree for the University of Maine.

The project is a food waste tracking system developed for the Mitchell Center here at the University of Maine.

## 1.1 Purpose of This Document

This document provides a more detailed and technical description of the system under design. It includes sections on system architecture, detailed UML class diagrams and use case diagrams detailing product scope. This document also details data design, including database specifications and structure.

## 1.2. References

The Mitchell Institute has provided Kayak Development solutions with a Google Drive shared folder, with documentation on all of Solution 1, the food waste tracking portion of the Mitchell Institute Food Waste Reduction Product. Additionally, within this shared drive are logos and color scheme graphics that have proved useful for the aesthetics of this product.

The documents proven useful are

* FRM\_LogoSet
* Home Tracker
* Home tracker V2
* ColumnTrackers
* Useful Notes/Powerpoints
* Kayak System Requirements Specification
* Waste Watcher UI mock-up

All but the SRS and UI mock-up come from the Mitchell Institute shared drive and were named by members of the Mitchell Institute.

# **2 System Architecture**

This section describes the way our app will be built and its foundational design structures. This includes the architecture of our application and the interactions between front-end and back-end components. The hardware and software the system requires will be outlined and presented via illustrations with correlated descriptions of the technological components we have defined.

## 

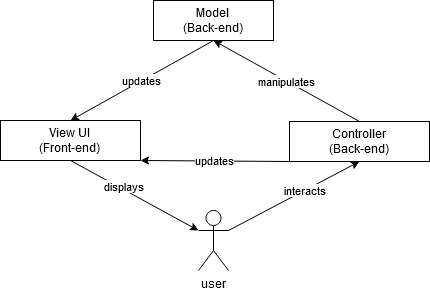
## 

## 

## 

## 2.1 Architectural Design

Model-View-Controller (MVC) Architecture Diagram: Figure 2.1



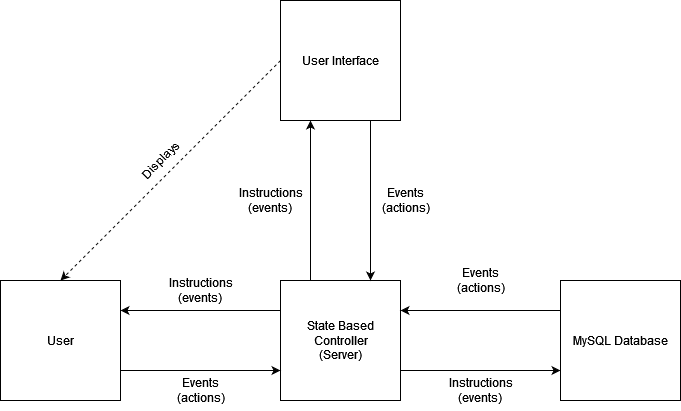
*Description: 2.1*

The mobile application must be able to be downloaded for both android and apple products. To do this the application architecture will consist of two main components, a back-end and a front-end. The Front-end, aka UI, will be created using React Native which is an open-source JavaScript UI software framework for mobile application development. This will allow the application to have a displayed interface that the user can interact with, hence the acronym “UI”. This includes buttons, pictures, graphs, text boxes for inputs, and various other visuals/interactive elements. The back-end is the database/server side. This is where user information and data will be stored and analyzed. To accomplish this, a MySQL database management system will be employed. It will be hosted on an Ubuntu Linux distribution server where data will be stored in a secure manner. To ensure user data protection, hashing will be implemented to secure user passwords and virtually all forms of sensitive user data gathered. All of these features will allow the mobile app to be downloadable cross-platform from both android and mobile apps.

**2.1.1 Technology**

An overview of the technology that the system will use will be discussed in this section. In regards to hardware as a whole, we have determined that we will be creating a mobile application which implies the need for a smartphone. Furthermore, we will need a server to host and manage our applications database, we are thinking of hosting through a form of UMaine infrastructure. ***(Still hashing this out, talk to Dr.Yoo before submission tomorrow)***. For software components of the system, we will be using a GitHub repository for storage and management purposes of the applications code. Visual Studio Code will be the primary tool for a programming environment across the team. Moreover, we will be coding the application using a JavaScript framework suited for mobile development for the front-end and mostly Java and some Python for the backend, both of which will be discussed thoroughly in sections to come. MySQL will also be used for a database management system. Apple and Android operating systems will need to be available for running the application as well as access to an Ubuntu Linux operating system for database hosting/management.

High-Level Event Driven Component Interaction Diagram: Figure 2.1.1



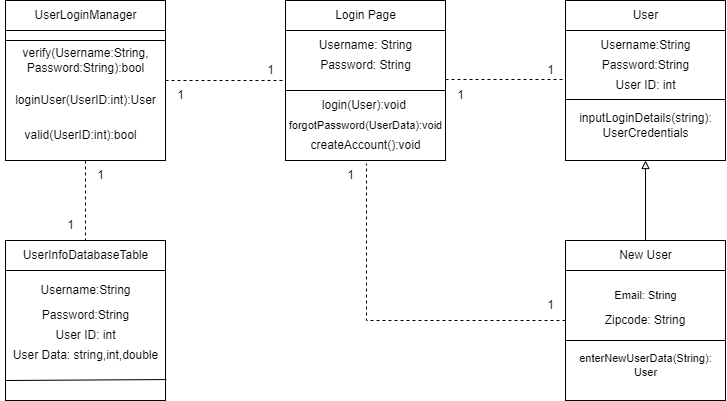
*Description: 2.1.1*

This is an alternate view of the previous diagram. More detail is involved including the MySQL Database. This diagram shows the flow of the technology involved in creating the architecture of the program as well as an overview of the interactions between one another. Overall, there are three components: the User, the UI, and the Database all of which take in instructions and respond with events. There is a Controller(Server) as well which takes in events from the components and responds with instructions. Events are only generated by components, an example of which would be the UI taking in an input, from which the event is then sent to the Controller. From here, the Controller receives and interprets the information it was given and formulates a set of instructions in response to the event. The Controller then sends those instructions to the specific components that need to be updated via the given event. Continuing from the previous example, the Server would take in the UI generated event and send an instruction to the Database component telling it to store the input given from the UI. The server would also send a similar instruction back to the UI telling it to update its display accordingly which is then shown to the user.

## 2.2 Decomposition Description

Below are two class diagrams detailing the main classes to be implemented in the application and their relationships. The application will be primarily object-oriented with the possibility of some functional programming.

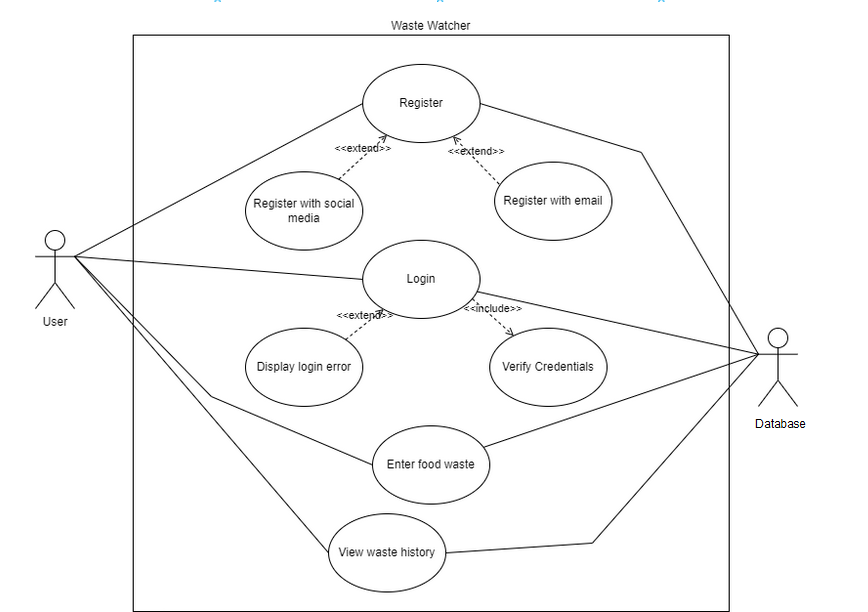
Login Functionality Class Diagram: Figure 2.2



*Description: 2.2*

The separate pages of the application will be object-oriented as each individual page will have its own specific functionalities/classes associated with it. Most pages will inherit class data identified in the login page above with the exception of the input food waste page. Thus, a detailed class diagram illustrating user account login and creation has been provided. Each box represents a class in which it contains its associated variables with their correlated data types (int, string, double). Each class also contains the functions/methods that it will be involved with. Any parameters passed to the function are found within the parenthesis and the functions outputs are found after the colon character. Outputs can be any associated data type based on the functions intended purpose. For example, in the New User class, the *“enterNewUserData(String):User”* method would take in a string for Username, Password, and Email and return a new instance of the User class. Dashed links between classes imply that there is a one-to-one relationship between the two. Essentially what this means is any singular instance of a class will have exactly one instance of a connected class associated with it. So, for the case of the Login class, there is only one User that can be associated

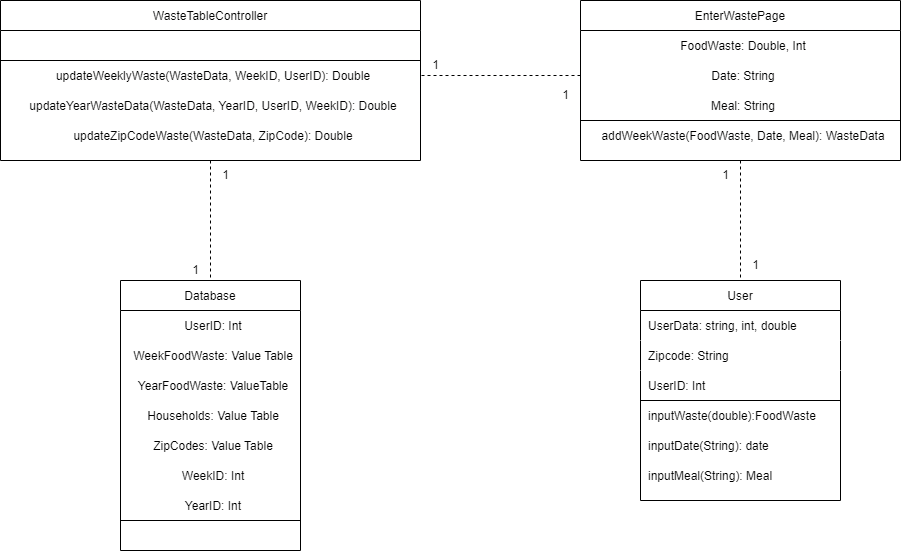
Use Case Diagram: Figure 2.2.1



*Description: 2.2.1*

This is a more general use case diagram. This mainly shows the core functionality and architecture of the program. This diagram includes named features a user could invoke.

Waste Input Functionality Class Diagram: Figure 2.2.2



*Description: 2.2.2*

This class diagram diagrams the food waste input functionality. The user interacts with the application, entering food waste for each meal on a specified day into the UI. The waste table controller class then takes this data and updates the weekly database table and yearly database table, adding the new data entered by the user to the respective totals. This data will be used to model food waste statistics at a later point in app development.

*Architecture Implications:*

The architecture our application falls under is the Model-View-Controller architecture. Model-View-Controller is named such due to the three main components involved. The model which represents the back-end database, the view which represents our applications User Interface, and the controller which represents our app controller. The controller’s main purpose is to communicate with the database and fetch data to display to the user or store data that the user has entered.

The most important interaction in a Model-View-Controller architecture is with the user interface and the user. Without being able to see the response from their inputs the entire system breaks down. The controller must also take user input into account and reflect this on the user's display.

All data must be stored appropriately in our database, but more specifically in the specific table that data is organized to. We access this data structure by performing requests from the controller that prompt the database to acquire specific data by determining where to look for it. Its output is managed by the controller and more than not displayed back to the user.

# **3 Persistent Data Design**

This section describes the system database including operating system information, database type, a database schema diagram and descriptions. Section 3.2 also outlines the database’s file structure if applicable.

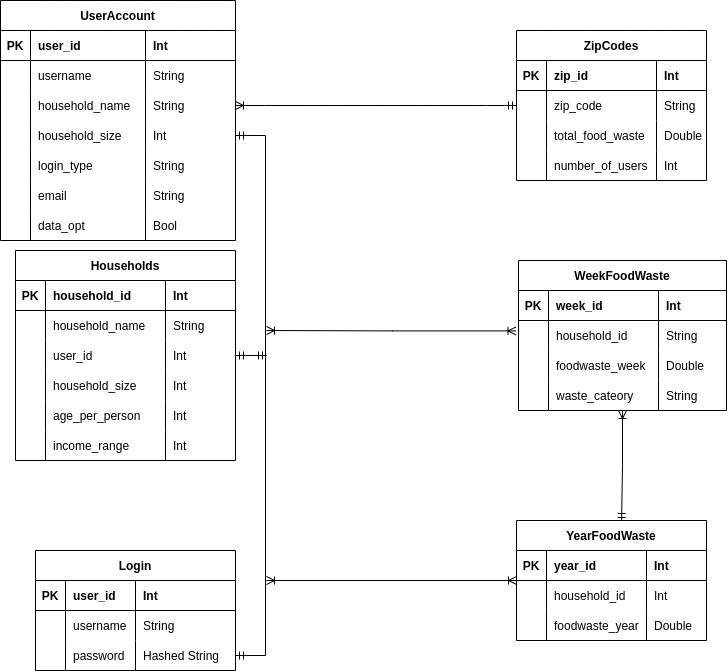
## 3.1 Database Descriptions (if you use a database)

We will be using a MySQL database hosted on an Ubuntu server to store user’s data. The system will collect the following data:

* Household name (String)
* Username (String)
* Password (String)
* Household size (int)
* Zip Code (String)
* Household income range (int)
* Amount of food wasted per meal (float)
* Number of household members (int)
* Age Range(String)
* User Opt In/Out of Data Collection (Boolean)

The data will be organized in the database as detailed by the entity relationship diagram shown below, with each box representing a different table within the database. Data collected about the user which is more sensitive such as income range, ages of members in the households and passwords, are stored in separate tables in order to add a level of obfuscation. Passwords stored will also be hashed for another level of security. For our physical infrastructure, we plan to host the Ubuntu server using physical machines at the University of Maine.

Database Entity Relationship Diagram: Figure 3.1



*Description: 3.1*

Each of the tables in the diagram above represent a table within our MySQL database. The UserAccount table will store information given to the UI when a user registers for an account on the application. The Login table also stores user specific information, specifically the username and password. The Household table stores more granular household information, including the age of each person in the household and the income range of the household. As stated above, information which is user/household specific is split into multiple tables, linked with the user\_id data in order to improve the security of the database. The ZipCode table will compile user waste data input by users to provide a breakdown of food waste data within a specific geographic area. The WeekFoodWaste table will be a compilation of all food waste submitted by users each week. Entries will be added to the table on a weekly basis, and the total food waste will be updated once users start inputting data. Meal specific food waste data (i.e how much food a household wasted at dinner on a specific day) will be collected and stored locally on the user’s device, and deleted once it is added to the broader database table. Lastly, the YearFoodWaste table takes data from the WeekFoodWaste table and will create entries on a yearly basis, tabulating total waste figures when weekly data is collected. All entries in these tables will be given a respective identifier which will be used to link data within different tables.

## 3.2 File Descriptions

No files are required for this system.

# **4 Requirements Matrix**

This section contains a table referring to our system Use Cases and the corresponding functions/methods that will perform the operations that satisfy the use case.

Requirements Matrix: Figure 4.1

| Use Case No. | Name | Function/Method? |
| --- | --- | --- |
| 1 | Input Food Waste | TrackWaste() |
| 2 | Create a Login | CreateLogin() |
| 3 | Input Demographic Information | StoreUserInformation() |
| 4 | Send Referral Message | SendMessage() |
| 5 | Opt Out of Data Collection | OptFlag() |
| 6 | Language Support | LanguageOpt() |
| 7 | Leaderboard Management | Leaderboard() |
| 8 | Badge system, and achievements | Achievements() |
| 9 | Tips for users | Tips() |
| 10 | History tracking | HistoryStats() |
| 11 | Trend of a users waste (day to day) | Trends() |
| 12 | Total food cost | TrackWasteTotal() |
| 13 | statistics (community) | CommunityStats() |
| 14 | statistics (state) | CommunityStats() |
| 15/16 | home/away eating habits | Habits() |
| 17 | additional info/resources page | HelpPage() |

# Appendix A – Agreement Between Customer and Contractor

By signing this document you agree that the requirements and specifications about the applications are good to pursue. These requirements are susceptible to change based on if new more important requirements come up or if things become obsolete. In the future, if these changes need to be made, they will be discussed with our customer first for clarification and then updated here once an agreement has been made. Changes to this document should be known by whoever it is relevant to.

**Kayak Development**

SIGN Declan Brinn X

SIGN Chase Pisone X

SIGN Finn Jacobs X

SIGN Levi Sturtevant X

SIGN Gavin Palazzo X

**Solution 1 Mitchell Institute**

SIGN \_\_\_\_\_\_\_\_\_\_\_X

SIGN \_\_\_\_\_\_\_\_\_\_\_X

SIGN \_\_\_\_\_\_\_\_\_\_\_X

SIGN\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_X

# Appendix B – Team Review Sign-off

By signing below members have indicated that they agree to the information mentioned above in this document. You agree that this document may see changes in the future. There should be no major disagreements, only minor disagreements.

SIGN Declan Brinn X Comments: None

SIGN Chase Pisone X Comments: None

SIGN Finn Jacobs X Comments:

SIGN Levi Sturtevant X Comments: None

SIGN Gavin Palazzo X Comments: None

# Appendix C – Document Contributions

Levi: 20% Worked on section 4 with Chase, Section 1 updates, tied MVC architecture to project, 2.2 Architecture description, Section 3 introduction

Finn: 20% Created Use Case diagram and descriptions for architecture

Declan: 20% Created diagram and description for data design section and one of the class diagrams

Chase: 20%, Created diagrams and descriptions, completed tables, appendix a, b and c, and edited

Gavin: 20%, Created diagrams and descriptions for architecture and introduction section