**CS673 Software Engineering**

**Team 4 - MyDietHub**

**Software Design Document**

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Ray Clark | Team Lead | *Ray Clark* | June 14, 2021 |
|  |  |  |  |
| Chen Zhao | Requirement leader | *Chen Zhao* | May, 31, 2021 |
| Huimin Diao | Configuration leader | *Huimin Diao* | May, 31, 2021 |
| Danny Graziano | Design & Implementation Leader | *Danny Graziano* | June 14, 2021 |
| Josephine But | QA Leader | *Josephine But* | June 13, 2021 |
| Zixia Zhou | Security Leader | *Zixia Zhou* | June 14, 2021 |
|  |  |  |  |

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
| 0 | Group | May 18, 2021 |  |
| 1 | Group | June 1, 2021 |  |
| 2 | Group | June 14, 2021 | Integrate reg/login, usda api and CRUD code |

**Table of Contents**

[Introduction](#_87t9hln2vjz0)

[Software Architecture](#_buttcq9i221r)

[Design Patterns](#_x18fj36s1121)

[Key Algorithms](#_mtfbusfb0eq3)

[Classes and Methods](#_7ucksmkf6rzx)

[References](#_15tmymhipvdv)

[Glossary](#_8n34lvocupub)

# **Introduction**

This document provides the details of the software design for the **MyDietHub** software application. **MyDietHub** is a full stack application built on modern frameworks that allows a user to quickly compute the calories of a given food type and for an entire meal. The purpose of the application is to allow a user to track their caloric intake and thereby track and regulate their diet.

## Application Overview

According to data from the National Health and Nutrition Examination Survey (NHANES, 2013–2014 Survey), approximately 2 in 3 adults in the U.S. are either overwieght or obese (BMI > 30). Since diet control (calorie intake control) is one of the most important ways to fight obesity, our group will build an app that helps people calculate the calories of the food they eat and will provide tips, encouragement and warnings to help people fight obesity. Any person hoping to regulate their caloric intake is a potential user.

## Software System Design Goals

In developing our software system design, we will focus on an architectural structure that will satisfy the stated system requirements. Second, we will attempt to implement a design that will be flexible and enable us to make changes as necessary. Lastly, we will integrate various software qualities such as usability, reliability, scalability and security.

More specifically, the ‘External Quality Characteristics’ that we will integrate include the following:

1) Efficiency – Measured by response time and throughput

2) Reliability - Measured in terms of mean time failure, i.e., up-time

3) Robustness - System error handling

4) Usability – UI/UX, ease of use

5) Security – Integrating measure to protect against malicious attack

The ‘Internal Quality Characteristics’ will include the following:

1) Maintainability – Ease of revising and/or enhancing the system

2) Re-usability – Use OOD to ensure reuse of code

3) Readability – Integrate clear design structure

4) Portability – System must work across different platforms

## Class Relationships

Cohesion and coupling can help measure the dependency and complexity of the software architecture. Cohesion measures dependency within the subsystem, meaning classes with similar functionality should be grouped together, while Coupling measures dependency among subsystems. A low level of coupling implies that a change in one subsystem does not affect any other subsystem.

We will strive to obtain a high level of cohesion and a low level of coupling.

The subsystem can be further decomposed into finer-grain sub-subsystems. Usually, the number of top-level subsystems is 3-5 with each sub-system focusing on a single concern.

# **Software Architecture Overview**

Our overriding goal with regard to software architecture is to decompose the software system into a number of smaller subsystems (components) to reduce system complexity while allowing for change.

The subsystems or components are structured or decomposed based on functions and data flow. For example, as our system tracks meal calories, the system will feature a “Meal” class that will maintain the meal’s key attributes and functions such as adding a new meal. There will also be an external data class to access the USDA data through an API and a view class that includes the various UIs.

More broadly, our architecture can be divided into “frontend” and “backend” components, along with a data persistence layer. The frontend components include the user login and registration portal, calorie display page, calorie history page, and anything that generally displays or retrieves information to/from the user. The data that’s collected is used by the backend components of our app to perform the processing and data retrieval tasks, at which point the data will be returned to the frontend for display to the user. Callback functions will allow backend functionality to be accessed via frontend elements (such as a button click). Generally, the classes that will comprise the backend are *Meal, MealDatabase, FoodItem, USDACalDB*. The frontend classes are *User, Main User GUI, MealInputPage (and their respective subclasses)*.

## Layered Structure - Overview

In a layered architecture, the system is divided into layers, with each layer providing a related set of services to upper layers.

Layered architecture is a generic architecture that defines the relationships between components, not the roles of each layer, and the MVC pattern is a special type of the layered architecture. When the layers are distributed on physically separated nodes, the structure is referred to as a “Tiered Architecture.”

## MyDietHub - Layered Structure Architecture

Whereas, the MyDietHub app uses an MVC structure, it is also layered. The following diagram illustrates the layered nature of the architecture, along with how the various components interact with each other. Note that, whereas, we currently are hosting locally, for the final iteration the app will be hosted externally on the Heroku server.

## 

## 

## 

## Model-View-Controller – Overview

Model–view–controller (known as MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is accepted from and presented to the user.

This pattern has become very popular for designing web applications and there are several programming languages and frameworks, such as Django, Flask and Node, that facilitate implementation of this pattern.

In the MVC architecture, the model is responsible for application-domain data. The view displays the model and is often implemented using templates to render HTML pages. The controller (logic) handles incoming requests received from the browser and then passes the proper actions to the model and then returns the response to the user.

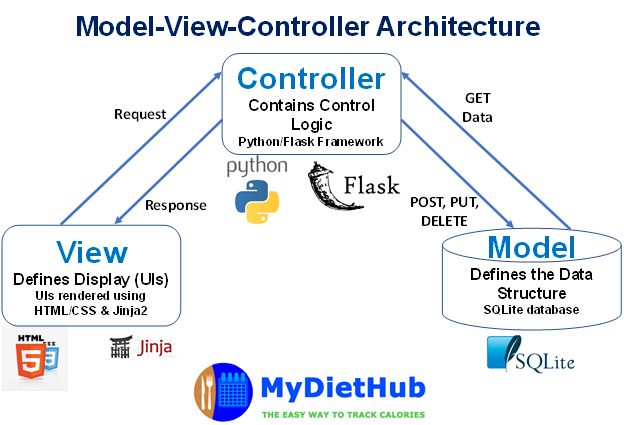
Decoupling the model and the view ensures that the presentation is always independent of application data. Decoupling the view and the controller ensures that the request made by the user can be independent of the resulting pages.

## MyDietHub Architectural Structure

For the **MyDietHub** application, we are using a traditional MVC layered architecture pattern. The frontend user interfaces (i.e., **View Layer**) are all developed using a combination of HTML5/CSS3 and Jinja2, which works with the Python-Flask API endpoints to handle user requests. All communications, i.e., requests and responses, between the user interfaces and the backend are channeled to various API endpoints for processing. For example, upon a button click, data from a form will be directed to the appropriate API endpoint or “route” where the code for handling the request is located.

The backend (i.e., **Control Layer**) for the application is built as a REST API using the Python-Flask framework. As discussed, it provides different endpoints to perform the various CRUD (create-read-update-delete) operations such as a POST request that will add a new meal to the database or a GET request that will return the contents of the meal to the user where the information is displayed on screen.

The persistence layer (i.e., **Data Layer)** in this application is implemented using a SQLite relational database (“DB”) to persistently store data. It is stable, cross-platform, and backwards compatible. It implements a small, fast, self-contained, high-reliability, full-featured SQL database engine. As noted, the Python-Flask API backend performs all DB operations by connecting to the DB and processing requests and responses.

Although we are currently hosting the app on a single local server during the development stage, we will ultimately migrate the application to an external Heroku server, which also contains the SQLite database. Hence, considering that the view layer will be rendered on a client machine, we will ultimately maintain a two-tiered structure. The diagram below provides a graphic illustration of our MVC architectural structure.

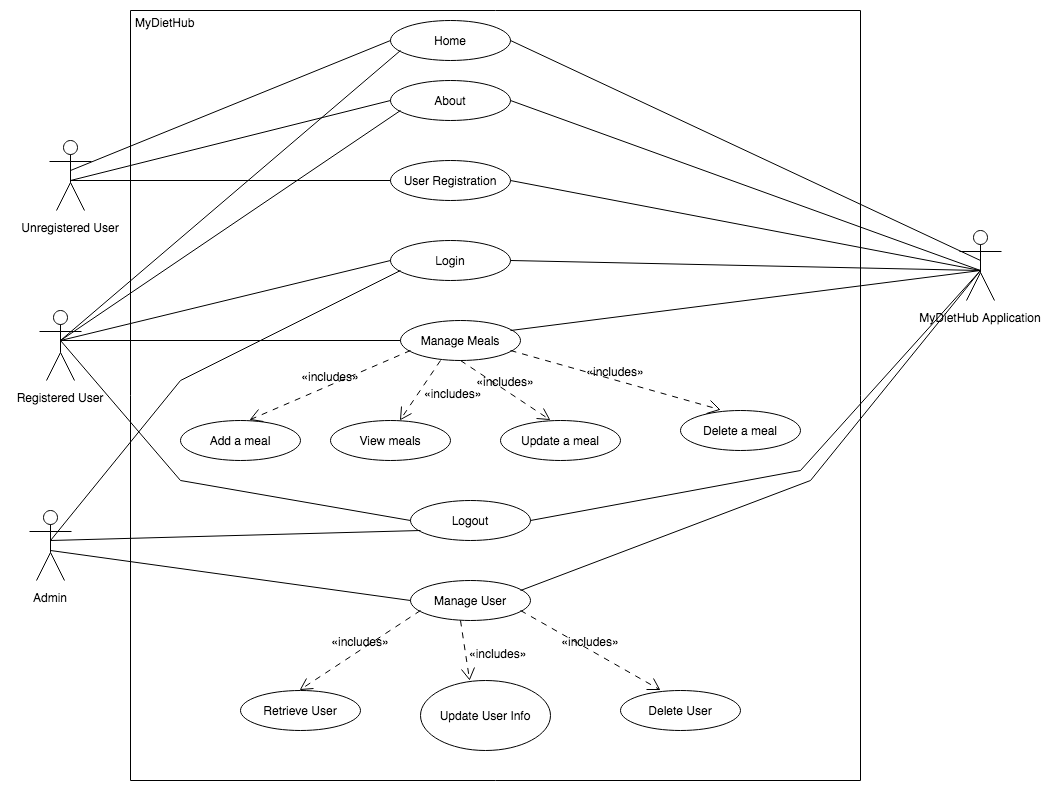
## MyDietHub Use Case Diagram

The use case diagram below outlines how users will perform tasks on **MyDietHub** application.

It outlines from a user’s point of view, a system’s behavior as it responds to a request.

Unregistered User can navigate to “Home”, “About” and “User Registration” to do registration.

Registered User can navigate to “Login” page to login to manage his/her meals. Admin can login to the system as administrator to manage the user information.



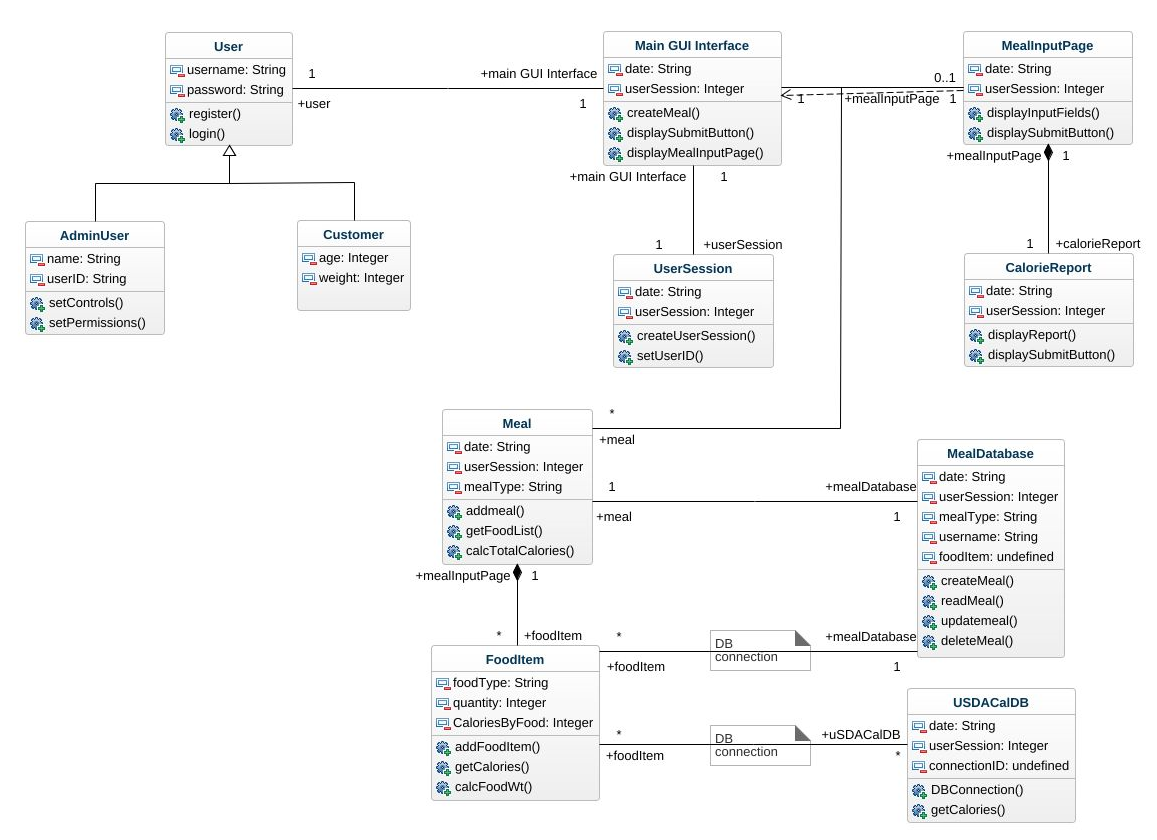
# 

# 

# 

# 

## MyDietHub Class Diagram

The class diagram below outlines the key classes, attributes and functions for the **MyDietHub** application. As discussed, the components are decomposed based on functions and data flow. In general, the classes can be grouped together into four (4) distinct packages, including: 1) User related classes, 2) View or GUI related user interfaces, 3) Model, or key object classes that store the food and meal data, and 4) Database (control) classes that are responsible for retrieving data from the user and 3rd party data sources and storing that data in a persistent structure. By employing this structure, we can achieve a high degree of cohesion with a low degree of coupling. Please note that, whereas, we have listed key fields and methods, not all are listed.

# 

# Database Design

### Current Table Schema Summary

#### User Model Table

|  |  |  |
| --- | --- | --- |
| **Column name** | **Variable type** | **Unique** |
| id (primary key) | String | True |
| firstname | String | False |
| lastname | String | False |
| email | String | True |
| username | String | True |
| password | String | False |
| role | String | False |

#### Meal Model Table

|  |  |  |
| --- | --- | --- |
| **Column name** | **Variable type** | **Nullable** |
| id (primary key) | Integer | True |
| meal\_type | String | False |
| food\_item1 | String | False |
| food\_item2 | String | False |
| calories | Float | False |
| date\_created | Datetime | True |

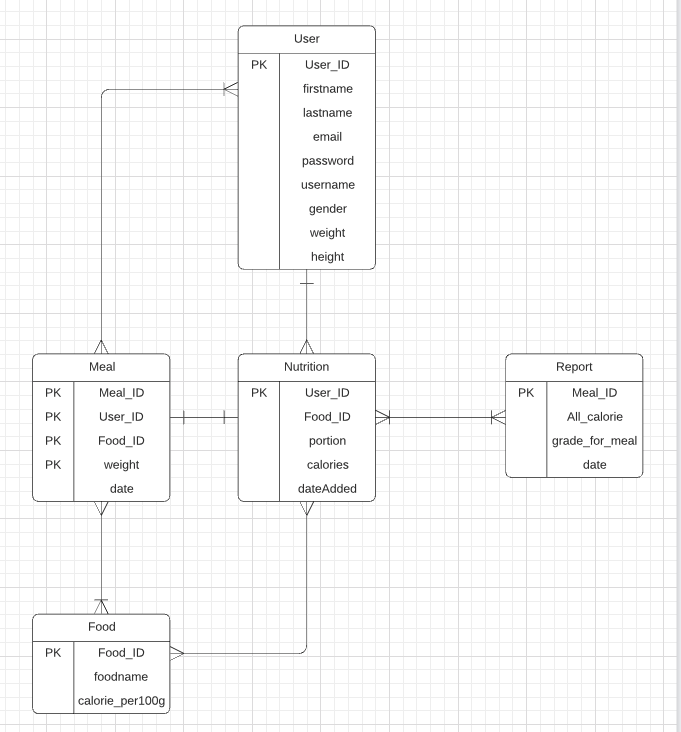
We will implement sqlite databases along with sqlalchemy to store both user and retrieved nutritional data . The sqlite/sqlalchemy combination works well with python3 and flask and there are many examples available of various use cases. Due to the time-constraints on this project, we elected to go with the most simple solution possible to form a minimum-viable product. Mysql is very similar to other sql-like databases, and can be replaced later on with a more advanced, but compatible database solution like Postgres.

Our “**User**” database will have the following keys: id, firstname, lastname, email, username, password. So that the application could calculate BMI and the appropriate calorie intake for each user.

Because the MyDietHub app needs to calculate the calories for each meal, so that “**Food**” can make sure the MyDietHub could find the calorie for each kind of food. And “**Counter**” can calculate the calorie for each meal with given data.

Our “nutrition” database has the following keys: userID, meal,

food, portion, calories, dateAdded



“**Report**” will retrieve relevant slices of the databases based on meal, calorie info and date range for eventual rendering and display to the user.

# **Security Design**

We have to take care of the user password before storing it into the database for security purposes. We use Flask-login, Werkzeug and we are going to use the method PBKDF2 (*method='pbkdf2:sha256')* to do the password hashing. **PBKDF2** is a simple cryptographic key derivation function, which is resistant to [dictionary attacks](https://en.wikipedia.org/wiki/Dictionary_attack)  [and rainbow table attack](https://en.wikipedia.org/wiki/Dictionary_attack)s. It is based on iteratively deriving **HMAC** many times with some padding. The **PBKDF2** algorithm is described in the Internet standard [RFC 2898 (PKCS#5).](http://ietf.org/rfc/rfc2898.txt)

We also take care to ensure that API keys we used to fetch the data from USDA are not exposed in our public code repository. They will be held in a separate configuration file that the user will populate with their own key.

**Design Patterns**

Design patterns provide a shared vocabulary to solve common problems. The famous design patterns book “Design Patterns: Elements of Reusable Object-Oriented Software” published in 1994 by the “Gang of Four” outlines 23 design patterns in three categories:

1) Creational patterns: design patterns about class or object creation, including abstract factory, builder, factory method, prototype, singleton, etc.

2) Structural patterns: design patterns about class and object composition, including adapter, bridge, composite, decorator, facade, proxy, etc.

3) Behavior patterns: design patterns about class or object communication, including command, interpreter, observer, state, strategy, template method, etc.

**Factory Pattern** – The ‘Factory’ pattern is one of the most used design patterns in software. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. With the Factory pattern, we create an object without exposing the creation logic to the client and refer to the newly created object using a common interface.

Given that a particular user of the calorie tracking app will most likely be creating meals of different types such as breakfast meal, lunch meal and dinner meal, and, perhaps, a mid-day or late-night snack meal, the Factory pattern may be particularly useful as all of these types of meals share common attributes and functions that can be implemented from as abstract class within the concrete sub-classes.

**Singleton** – The ‘Singleton’ pattern is one of the simplest design patterns. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. This pattern involves a single class which is responsible for creating an object while making sure that only a single object gets created. This pattern is particularly useful when creating a single user session with an associated ID. Hence, we may utilize this pattern for user session instantiation.

# **Key Algorithms**

Key algorithm

[Essential]

1. BMI calculator

We have user data so that we can get the user’s BMI,

USER BMI = weight(kilogram) / height^2 (meter)

2. Calorie calculator

We have the type of food and the weight of food which is input by user, so that we can get the calorie of each kind of food:

Each food calorie = [weight(gram)/100] \* calorie\_per\_100gram(for each kind of food)

3. Meal calorie

We have each food calorie in the second algorithm, so that we can get the sum calorie for one meal:

Meal calorie = sum(each food calorie)

# **UI Design**

This section presents the graphical user interface of the MyDietHub Web application, where all functionalities are grouped visually and logically according to the types of entities they are associated with.

The design foresees the development of the web application using the following HTML templates:

* HTML5;
* css;
* use of web standards

The section that follow contain mock views of a selection of the most important aspects

of the GUI, along with textual descriptions of their purpose and contents.

1. **Home (Landing Page)**

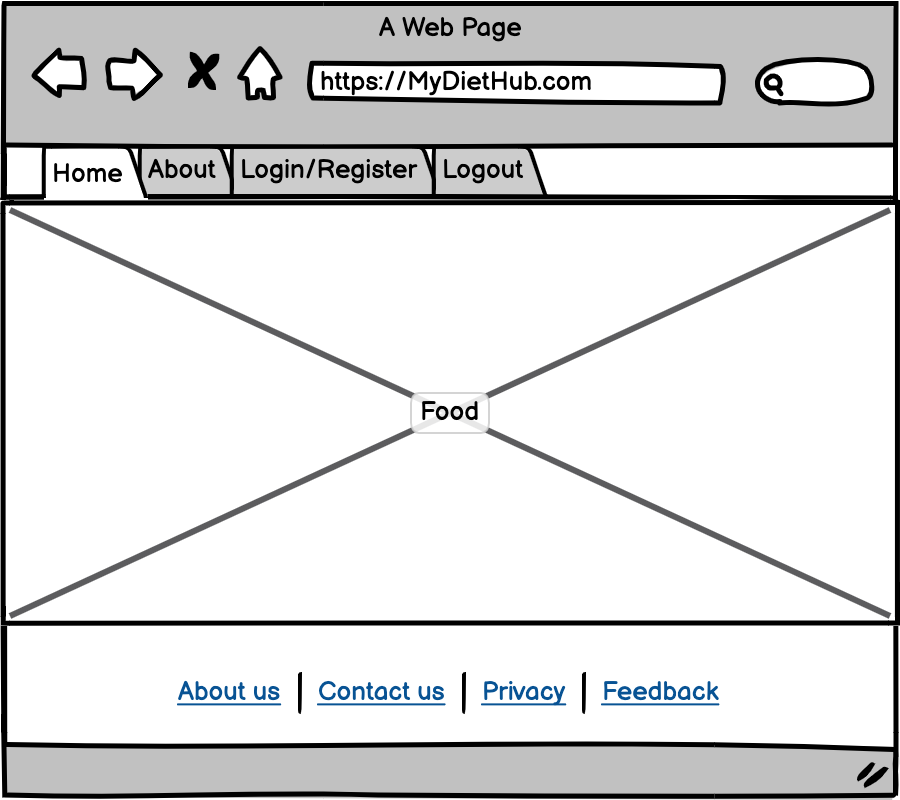
This section describes the “Home” page. It is a gateway to a site’s content. The main component is that the user can take a look on the “About” page, do the registration and login into the system.

Purpose :

A gateway to MyDietHub's content.

Navigation & User Interaction :

* User can click on “About” link to advance to the about page.
* User can click on “Login/Registration” link to advance to the login and Registration page.
* User can click on “logout” link to advance to the Logout page.



1. **About**

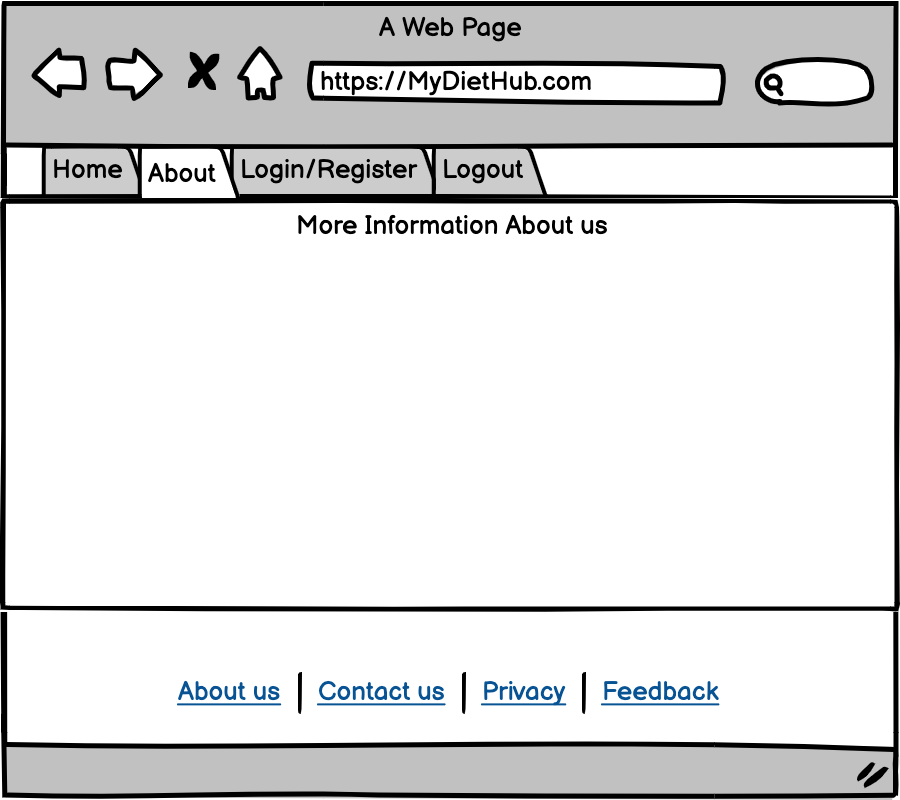
This section describes who we are for a visitor to learn more about us.

Purpose :

For the user to know more about us.

Navigation & User Interaction:

* User can select “Home” to go back to the landing page
* User can click on “Login/Registration” link to advance to the login and Registration page.
* User can click on “logout” link to advance to the Logout page.

****

1. **Login & Registration**

This section describes the “Login and Registration” page. The main component is the login and registration form. Users can login to their dashboard or switch to the registration page when they need an account. For the login part, where the user needs to provide the username and password. For the registration part, where the user needs to provide the firstname, lastname, username, password and the email.

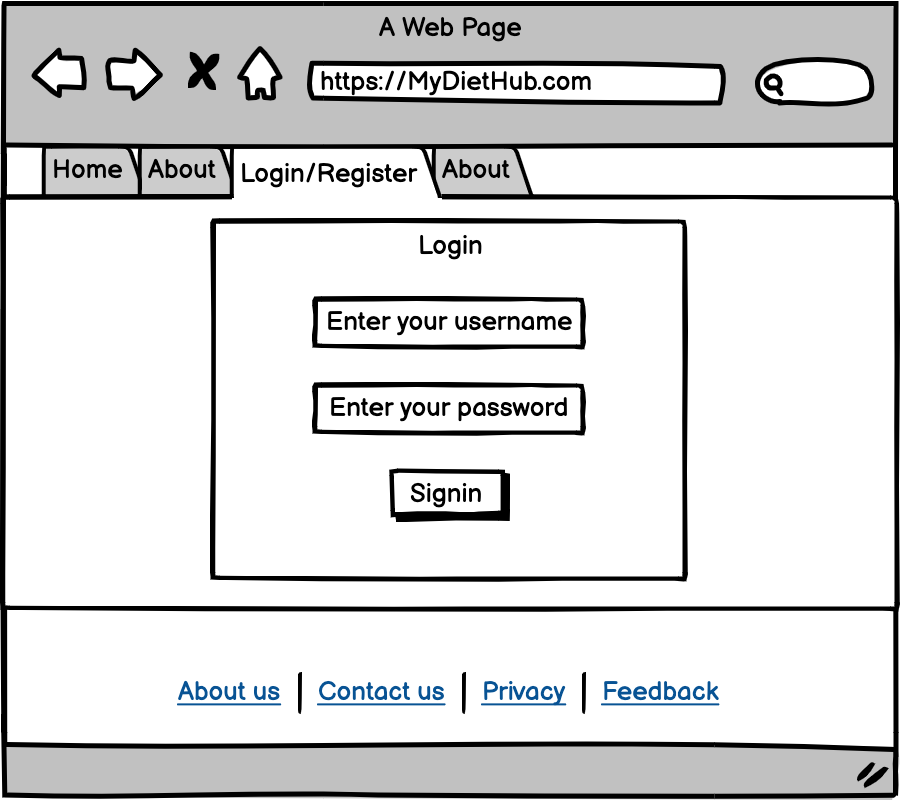
Purpose :

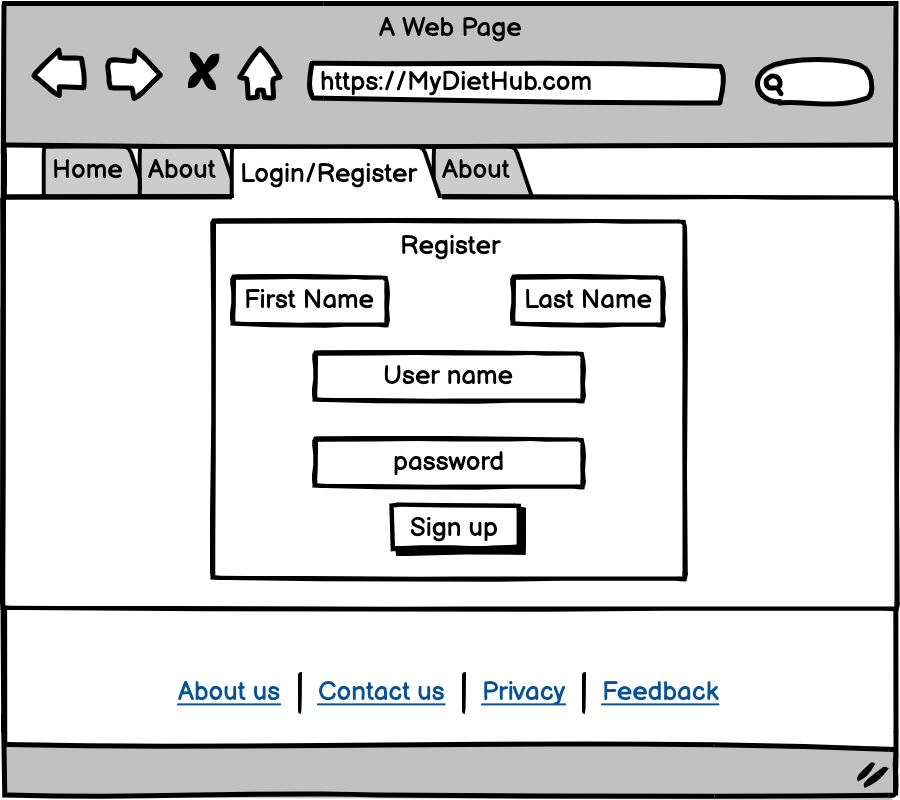
For the user to do the login and go to the user’s dashboard.

For the user to do the registration.

Navigation & User Interaction :

* To login the user is required to enter the user's username and password and click the “Sign In” button.
* The successful login will advance the user to the user’s dashboard.
* To register the user is required to enter the user's username, password and email and click the “Sign up” button.
* The successful registration will advance the user to the login page.





1. **User’s dashboard - Create Meal**

This section describes the user’s choice page. The main component is the “Create a Meal” button which lets the user navigate to the “Create a Meal” page.

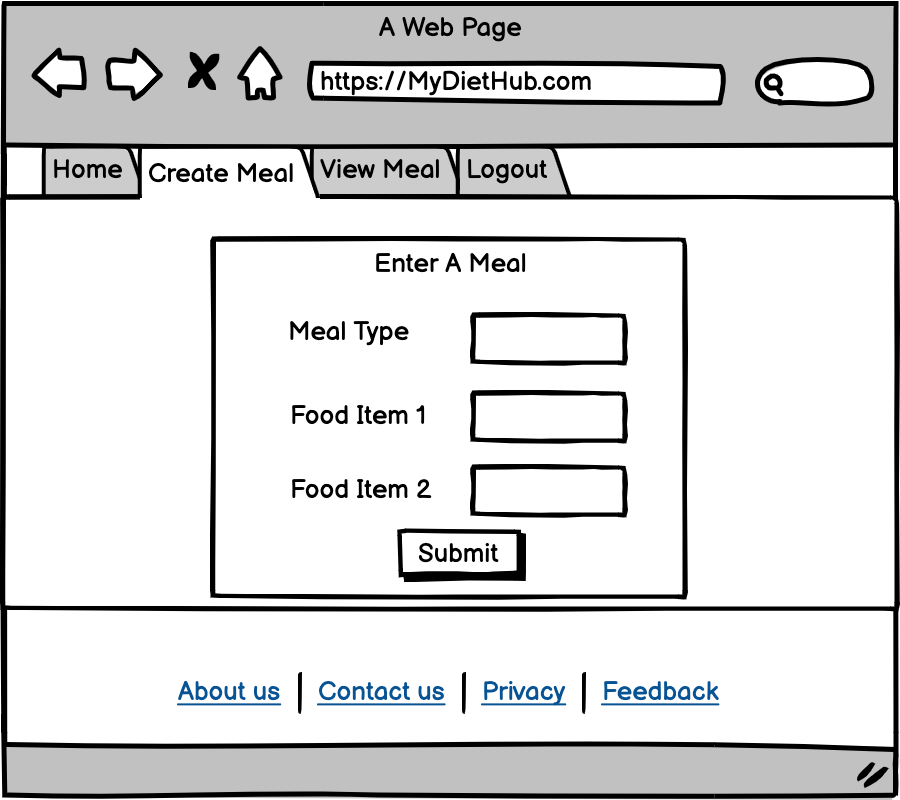
On this page, where the user needs to provide meal type and food items, after all inputs, the system will show the total calories and save this meal information. Also users can check it in “View Meal”.

Purpose :

For the user to go to “Create Meal”

Navigation & User Interaction :

* User can click on the “Create Meal” button to advance the user to the “Create a Meal/Calculate Calories” page.
* User can select “Home” to go back to the Home page.
* User can select “View Meal” to go to the View Meal page.
* User can select “Logout” to logout from the system and go back to the landing page.



1. **View Meal**

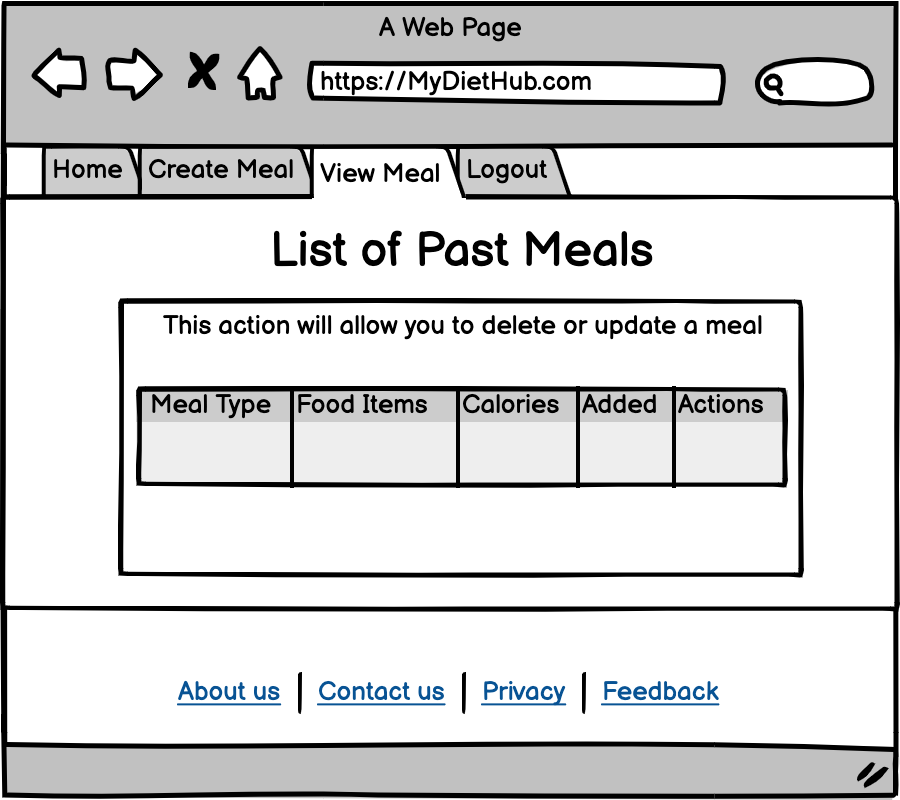
This section describes the “View Meal/Calculate Calories” page. On this page, where the user could see the meals that user input before, also user can see the “Meal Type”, “Food items”, “Calories”, “Added”- which is the time user added this meal,”Actions”- user can update the meal or just delete it.

Purpose :

For the user to see the calories of a meal, update the meal information or delete the previous meal information.

Navigation & User Interaction:

* User can click the “Home” link to advance to the Home page.
* User can click the “Create Meal” link to advance to the Create Meal page.
* User can click the “logout” link to advance to the landing page.



# Classes and Methods

Please provide a link to your application API document which should be automatically generated.

# References

# 1) <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

# Glossary