
## CSC 431

# IoT Application for Grocery

## System Architecture Specification (SAS)

**Team 19**


| |
|---|
| Joshua Welsh |
| MinA Jang |
| Andrew Burch |

# Version History

| Version | Date | Author(s) | Change Comments |
|---|---|---|---|
| **1.0** | April/9 | Josh | |
| **2.0** | April/20 | MinA | Edited documents according to the comment |
| **3.0** | May/4 | MinA | Addressed comment for the final submission |

# Table of Contents

# Table of Figures

# 1.System Analysis

## 1.1 System Overview

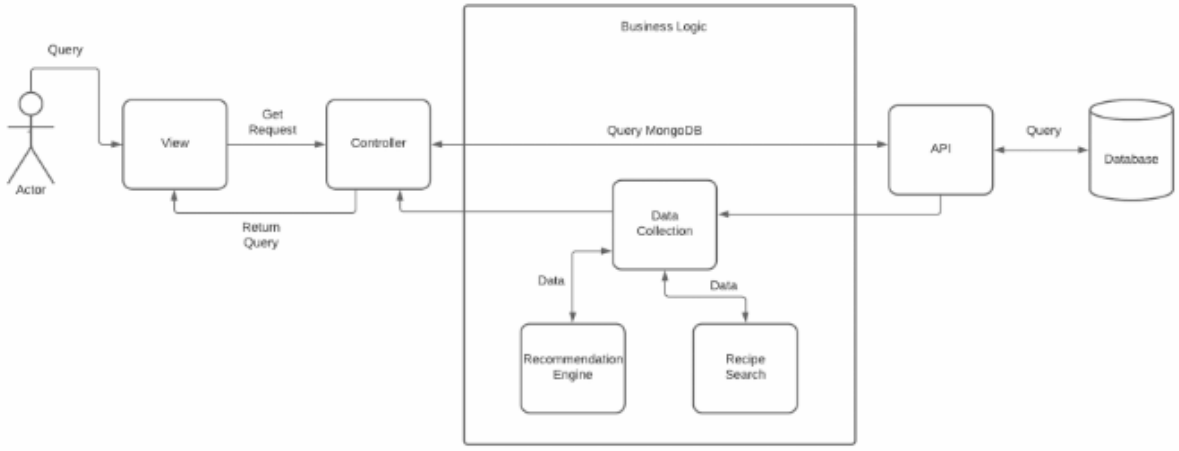
We will use a four-tiered architecture deriving from the MVC framework to build the IoT Groceries App. The MVC framework or Model Viewer Controller framework is a three-tiered architecture where the user's UI is separated from the database manager or model. The user experience is called the view. The controller acts as an intermediary between the view and model allowing the model to update the view and the view to alter the model. We will add a 4$^{th}$ tier between the controller and model to handle business logic such as recommendations and available recipes.

The user view will use React-Native and we plan to develop the controller as well as other server side capacities using Nodejs specifically the Express framework. Since business logic will involve machine learning we will build it using Python and Tensorflow and it will communicate to the controller using REST API endpoints. The model will be a noSql database using mongoDB.

A typical use case will involve the user accessing the app on their smart calling in the user view. The view query the controller for data such as current refrigerator stock and information about it. The controller will query the model and on return pass through the business logic layer. This layer will send data through private API endpoints to a python data archive and based on received user data, recommendations and recipes will be returned to the controller. The controller will then return the information to the view to complete it.

## 1.2 System Diagram

**FIGURE 1 USE CASE DIAGRAM**



## 1.3 Actor Identification

There is only one type of human actor – authorized users. They must download the app and have an account. If a user does not have an account, they must create an account.
The non-human actor is an API which retrieves the required data from the database.

## 1.4 Design Rationale

### 1.4.1 Architectural Style

Most computation done by this system can be handled on the backend or server side. This system also functions based on data retrieved from the database model. Due to these 2 factors, we use a 4-tier architecture to structure our system. It provides abstraction by separating the front-end rendering from the computationally intensive jobs done by communicating with the database and performing business logic.

### 1.4.2 Design Pattern(s)

We use the façade design pattern. User request prompts a function to call complex subsystems. For example, reading barcodes to store grocery data invokes subsystems of loading barcode data, matching the data according to the database, and then finally storing the processed data. Moreover, enabling expiration notification involves a subsystem of grocery timer to keep track of the expiration date, notification function to send alarms if the expiration date is coming up.

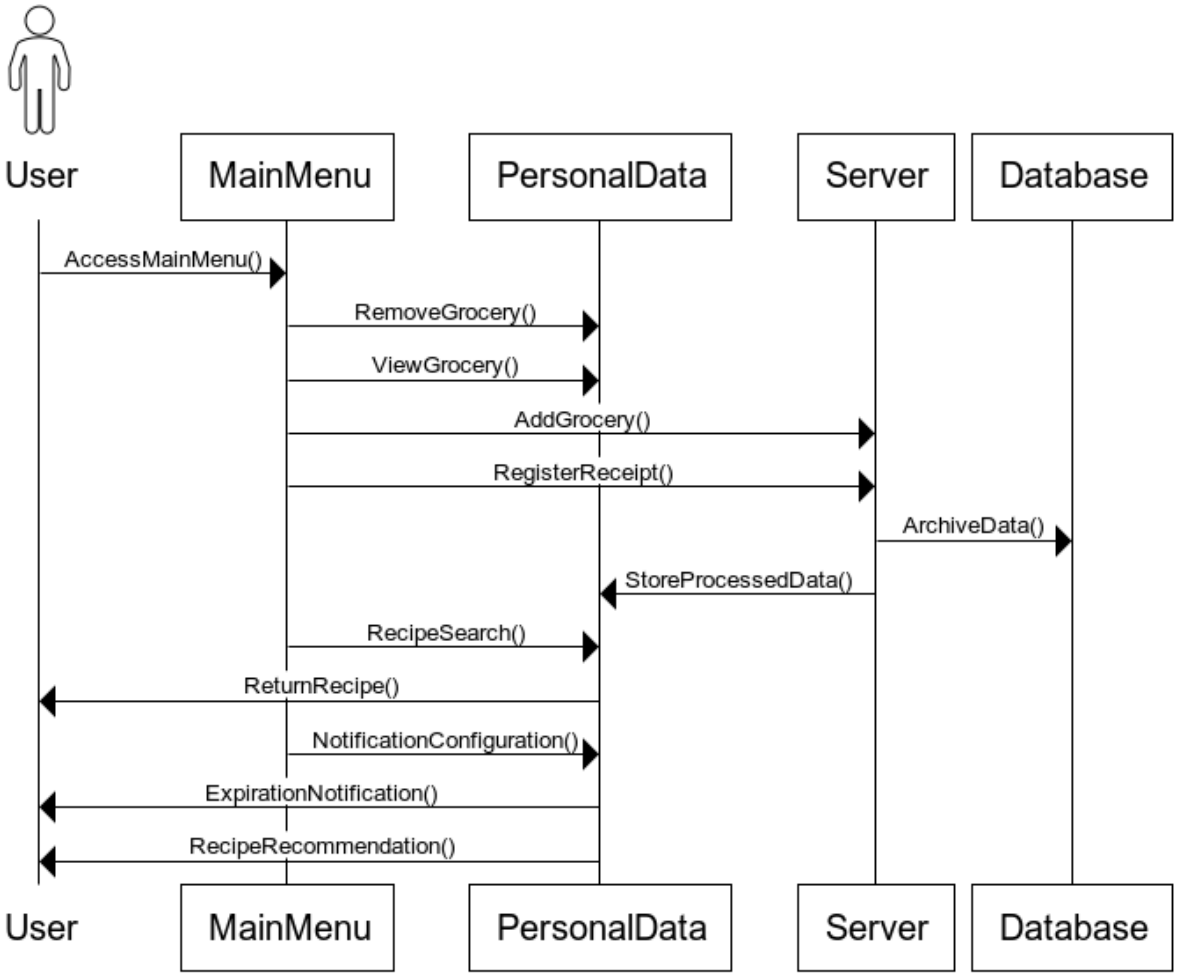
### 1.4.3 Framework

We utilize the MVC framework for our system. We give users access to modifying their grocery lists. Adding or deleting items will alert the controller of the request, controller updates the model which interacts with the database, and the model alerts the user view of the change, as the view updates itself by grabbing model data.

# 2. Functional Design

## 2.1 Sequence Diagram

**FIGURE 2 SEQUENCE DIAGRAM**



# 2.2 Structural Design

## FIGURE 3 CLASS DIAGRAM