IoT Based Smart Food Monitoring System for Fridge

This project is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering.

Arjon Das
ID: 1304016

Supervised by
Professor Dr. Mohammed Moshiul Hoque
Department of Computer Science & Engineering (CSE)
Chittagong University of Engineering & Technology (CUET)

Department of Computer Science & Engineering
Chittagong University of Engineering & Technology
Chittagong-4349, Bangladesh.
December, 2018

The project titled **"IoT Based Smart Food Monitoring System for Fridge"** submitted by ID No. 1304016, Session 2016-2017 has been accepted as satisfactory in fulfillment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering (CSE) as B.Sc. Engineering to be awarded by the Chittagong University of Engineering & Technology (CUET).

Board of Examiners

1._____ **Chairman**

Dr. Mohammed Moshiul Hoque **(Supervisor)**

Professor

Department of Computer Science & Engineering (CSE)

Chittagong University of Engineering & Technology (CUET)

2._____ **Member**

Dr. Mohammad Shamsul Arefin **(Ex-officio)**

Professor and Head of the Department

Department of Computer Science & Engineering (CSE)

Chittagong University of Engineering & Technology (CUET)

3._____ **Member**

Dr. Mohammad Shamsul Arefin **(External)**

Professor and Head of the Department

Department of Computer Science & Engineering (CSE)

Chittagong University of Engineering & Technology (CUET)

Statement of Originality

It is hereby declared that the contents of this project is original and any part of it has not been submitted elsewhere for the award of any degree or diploma.

Signature of the Supervisor **Signature of the Candidate**

Acknowledgement

First of all, I would like to thank the Almighty for successful completion of this project. The satisfaction that accompanies the successful completion of this project would be incomplete without the mention of people whose ceaseless cooperation made it possible, whose constant guidance and encouragement crown all efforts with success. I convey my humble gratitude to my respectful project supervisor Dr. Mohammed Moshiul Hoque, Professor, Department of Computer Science and Engineering, Chittagong University of Engineering and Technology, for his valuable advice and sincere guidance throughout my project work. I convey special thanks and gratitude to Dr. Mohammad Shamsul Arefin, Head of the Department of Computer Science and Engineering, Chittagong University of Engineering and Technology, for his encouragement and cooperation. I want to express my gratitude to all my respected teachers of the department. I would like to thank all my friends and the staffs of the department for their valuable help. Finally, I would like to thank my family for their constant love and support during my study period.

Abstract

Smart Food Monitoring System is an ecosystem of various hardware and software aiming towards automated monitoring of food or other edible object. Internet of Things is an open and complete network of intelligent objects that can be leveraged for extensive monitoring. Hence Smart Food Monitoring System based on IoT can provide verities of commercial, industrial and household benefits. The proposed system suggests systematic use of sensors to perform refrigerator environment quality and food quantity. The system can monitor refrigerator environment via temperature sensor, tell amount of inventory via weight sensor and produce visual feedback of the refrigerator storage using a camera housed inside. By harnessing the power the of IoT technology the system will enable the user to get notified when there is shortage of food or the fridge is not cooling the food enough in which case active user can take immediate preventive measures. Food management in homes as well as restaurants, and cold storages with the help of these of system that monitors the quantity and generates alerts, hence proactively controls wastage and inventory all through the comfort of users mobile device. The Smart Food Monitoring System will lookup for food at best efficiency and fast response to notify the user any where anytime. The system also stores valuable temperature data of long interval so that the user can examine the temperature curve of the refrigerator to see if the food is preserved at the right optimal temperature. The system can achieve these level of performance via dedicated server for the logics and modern sensor unit driven by Raspberry Pi micro controller and the data is sent to the user via user mobile application which is simple and intuitive to use for the purpose it is built for.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Introduction

The Internet of Things or IoT is the next frontier in technology. It is the ability for things that contain embedded technologies to sense, communicate, interact, and collaborate with other things, thus creating a network of physical objects. IoT is an open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources, reacting and acting in face of situations and changes in the environment. IoT is maturing and continues to be the latest, most hyped concept in the IT world. The IoT aims to unify everything in our world under a common infrastructure, giving us not only control of things around us, but also keeping us informed of the state of the things. Over the last decade the term IoT has attracted attention by projecting the vision of a global infrastructure of networked physical objects, enabling anytime, anyplace connectivity for anything and not only for any one [1]. IoT describes a world where just about anything can be connected and communicates in an intelligent fashion than ever before. In these era of technology advancement, everything requires monitoring and controlling. And harvesting the power of modern technology to monitor food can make a real difference in preventing food waste. By properly monitoring the foods in fridge it is possible to maintain the quality of the food to its best. Traditional fridges dont come with any sort of food monitoring system. Thats why users remotely dont get to know whats inside their fridge. Analyzing the quality of food storage environment by examining

temperature inside a fridge is crucial for better food quality. IoT technology can help such action to be more intuitive. Food monitoring has become essential these days as more and more food is being wasted just by storing them in excess in the refrigerator and thrown away when they are expired. These leads to both environmental and economical impact. Only if an automated system that can help to notify the inventories about scarce product, remotely monitoring the foods and assessing refrigeration performance can help to reduce food wastage. The idea of connecting home appliances through the internet is regarded as a phenomenal idea. Because if these home appliances can provide extensive functionality which can be accessed by anyone over the internet with proper authentication. Food management in homes as well as restaurants, and cold storage's with the help of these sort of system that monitors the quantity and generates alerts, hence proactively controls wastage and inventory all through the comfort of users mobile device. The system has to be intuitive so that the user can use the features at ease rather than feeling that the it is another gimmick.

1.2 Traditional Food Refrigeration System

Refrigeration has become an essential part of the food chain. It is used in all stages of the chain, from food processing, to distribution, retail and final consumption in the home. Foods tend to have a very little amount of time before degrading and losing the nutrient values but due to refrigeration of foods the chemical process of food degradation has been prolonged. Cold temperatures help food stay fresh longer. The basic idea behind traditional refrigeration is to slow down the activity of bacteria which all food contains so that it takes longer for the bacteria to spoil the food. The ideal temperature for food refrigeration is 4°C at which point micro organisms tend to slow down food breakdown process [2]. As the food degradation is prolonged due to the process less and less preservatives are needed for the foods, thus making them more healthy. Food refrigeration technique provides access to healthy food with high nutrition contents. These single innovation plays crucial role in provide healthy and nutritious food to people worldwide. Both the food industry and household refrigerator employs both chilling and freezing processes where the food is cooled from ambient to temperatures above

0°C in the former and between -18°C and -35°C in the latter to slow the physical, microbiological and chemical activities that cause deterioration in foods. It is possible to transform the traditional refrigerator into a smart food monitoring system by housing the sensor modules and proper internet connectivity. Figure 1.1 portraits the basic scenario who the system is embedded on a traditional refrigerator.

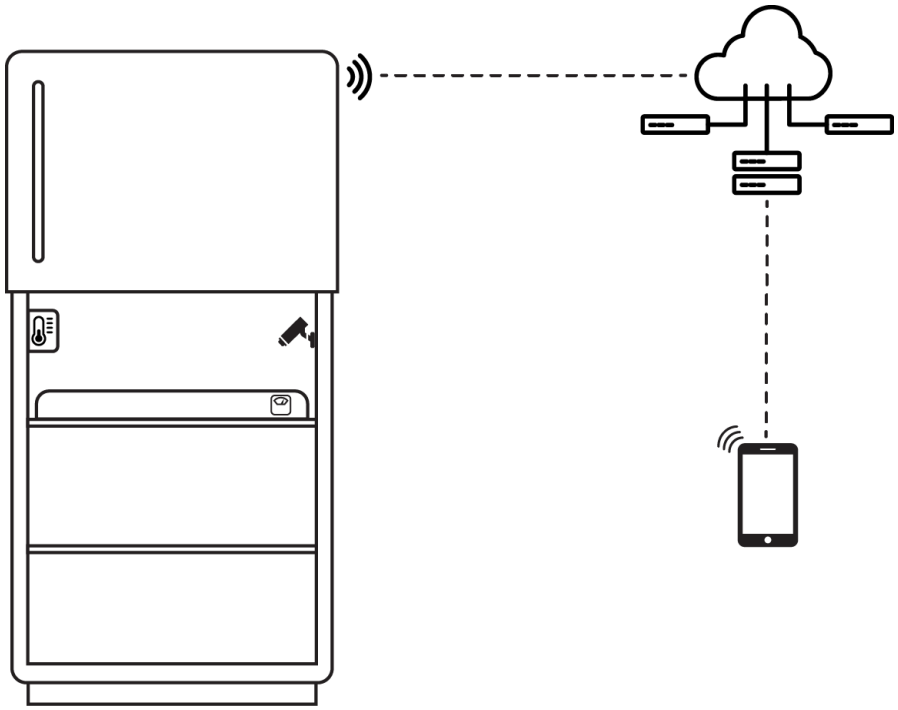

Figure 1.1: Smart Food Monitoring System embedded on a refrigerator

1.3 Motivation

Food refrigeration process is a quite mature process which has been shaped by the food industry for some decades so how can such system have any complication. Now a days one of the main problems addressed for refrigeration systems is actually wastage of food due to way more prolonged storage of food than they are intended to be. In developed countries up to 30 percent of household food ends in the bin, often after storing them in the fridge and misconceptions about food safety and exaggerated disgust. Sometime due to lack of extensive monitoring of the storage environment specially in the cold storage industries huge amount of food tends to rot for not getting acquainted by the cause. Again one of the main reasons of food ending up in the bin is lack of food supply knowledge or whats in

the fridge. So many consumers waste food because they dont know whats in their fridge during their grocery shopping and these problem further elevates because in most households there is usually one or two food buyers. Food refrigeration systems dont provide such get acquainted of these informations. In its current stage food refrigeration system needs to incorporate a monitoring system so that consumers can assess whats inside the fridge. So many refrigerators fail to maintain the optimal refrigeration temperature. Thats why the monitoring system should also provide an extensive view of the internal food storage environment so the consumer can assess the refrigerator performance because many food start to deteriorate rapidly due to bad refrigeration.

1.4 Challenges

Developing a IoT based Food Monitoring System can be challenging due to the fact that some of the components of the systems will be exposed to the low temperature environment of the refrigerator, which is initially fine but at sudden temperature rise that could cause water condensation on the electric components that can damage the system. Inventory monitoring via load sensor can be challenging as load sensors dont provide stable readings in noisy environments and often requires to calibrate the load sensors. Creating real time communication between the user application and the device application via server is also troublesome as it requires complex socket connections. And keeping the socket connections alive during user application idle mode is challenging because mobile operating systems tend to kill any sort of network connectivity when the application goes to an idle mode to conserve power.

1.5 Contribution of the Work

The main objective of this project is to develop a system to monitor foods in the refrigerators in real time based on IoT technology. The key objectives and possible outcomes of our project may identify as following:

- To assess the quantity of food and provide timely notification for restocking.

- To provide visual feedback of foods to user via mobile device when requested.
- To provide internal temperature monitoring feature to check optimal temperature.

1.6 Organization of the Report

The entire report is represented in six different chapters. We can outline the report structure as follows:

- Chapter 1 give an overview of IoT and refrigeration systems and discusses about the motivation behind this project. It also lists the objective of the project and challenges we may face in this work.
- Chapter 2 discusses about some technologies we have used in this project. It also give an overview of some previous works done in this sector and discuss their strength and limitation.
- Chapter 3 describe logical structure and architecture of our system.
- Chapter 4 describe implementation details of the system.
- Chapter 5 discusses the experimentation process and resulting outcomes of our system.
- Chapter 6 presents the conclusion and future research scope in this field.

Chapter 2

Literature Review

In these chapter we will discuss about general terms and technologies we used in our project. We will discuss some technical jargons and the techniques associated with the technologies. We will also discuss about previous work done in this field together with their merits and limitation.

2.1 Internet of Things

The Internet of Things is a novel paradigm shift in IT arena. The phrase Internet of Things which is also shortly well-known as IoT is coined from the two words i.e. the first word is Internet and the second word is Things. The Internet of Things can also be considered as a global network which allows the communication between human-to-human, human-to-things and things-to-things, which is anything in the world by providing unique identity to each and every object [3].

IoT describes a world where just about anything can be connected and communicates in an intelligent fashion that ever before. Most of us think about being connected in terms of electronic devices such as servers, computers, tablets, telephones and smart phones [4]. In whats called the Internet of Things, sensors and actuators embedded in physical objectsfrom roadways to pacemakersare linked through wired and wireless networks, often using the same Internet IP that connects the Internet. These networks churn out huge volumes of data that flow to computers for analysis. When objects can both sense the environment and communicate, they become tools for understanding complexity and responding to it

swiftly. Whats revolutionary in all this is that these physical information systems are now beginning to be deployed, and some of them even work largely without human intervention. The Internet of Things refers to the coding and networking of everyday objects and things to render them individually machine-readable and traceable on the Internet. Figure 2.1 portraits a basic model of IoT system.

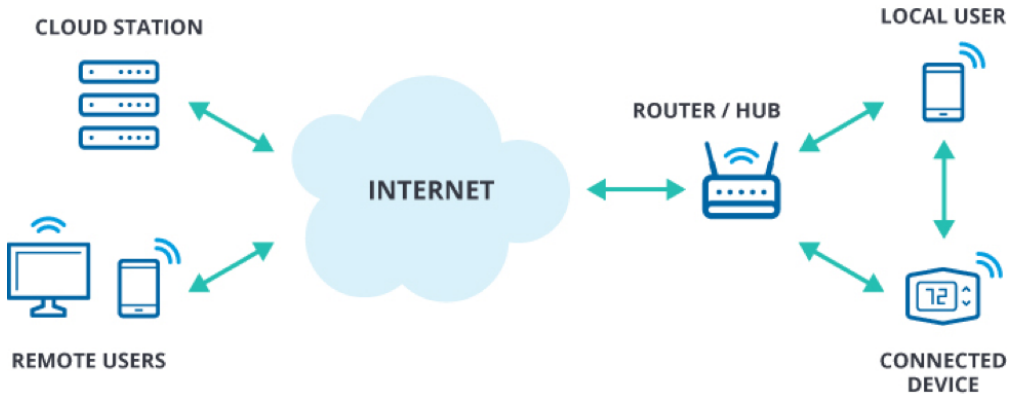

Figure 2.1: Basic IoT Model

2.2 REST API

A REST API also referred to as a REST web service is based on representational state transfer (REST) technology, an architectural style and approach to communications often used in web services development. It uses HTTP requests to GET, PUT, POST and DELETE data. The REST used by browsers can be thought of as the language of the internet. With cloud use on the rise, APIs are emerging to expose web services. REST is a logical choice for building APIs that allow users to connect and interact with cloud services. A REST API breaks down a transaction to create a series of small modules. Each module addresses a particular underlying part of the transaction.

A REST API explicitly takes advantage of HTTP methodologies defined by the RFC 2616 protocol. The API uses GET to retrieve a resource, PUT to change the state of or update a resource, which can be an object, file or block, POST to create that resource, and DELETE to remove it. REST API calls are stateless. Thats why it is useful in cloud applications. Stateless components can be freely

redeployed if something fails, and they can scale to accommodate load changes. By using stateless protocol and standard operations, REST systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running. [5]

2.3 Common Devices and Softwares for Smart Food Monitoring System

The design of IoT based Smart Food Monitoring System contains essential components that plays intricate role in the system architecture. These components carry out the logical execution, shows user interface and also gives the output to the user. Technologies and instruments used in our system are explained below:

2.3.1 Raspberry Pi

The Raspberry Pi is a series of small single board computer. The processor speed ranges from 700MHz to 1.4GHz for the Pi3 Model B+ [6]; on-board memory ranges from 256MB to 1GB RAM. The Raspberry Pi may be operated with any generic USB computer keyboard and mouse. It may also be used with USB storage, USB to MIDI converters, and virtually any other device or component with USB capabilities. Raspberry Pi has 40 dedicated interface pins. In all cases, these include a UART, an I2C bus, a SPI bus with two chip selects, I2S audio, 3V3, 5V, and ground. The maximum number of GPIOs can theoretically be indefinitely expanded by making use of the I2C or SPI bus. A Raspberry Pi is shown in Figure 2.2

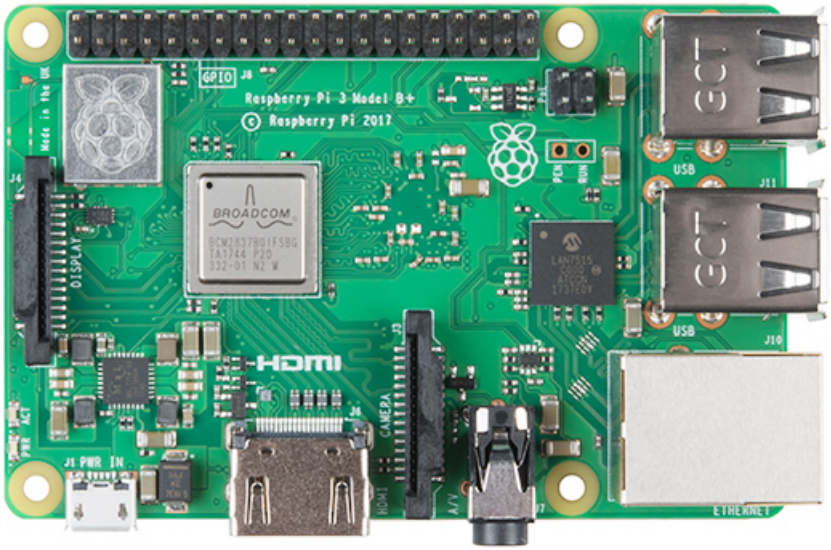

Figure 2.2: Raspberry Pi 3 B+

2.3.2 GPIO Ports

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all Raspberry Pi boards. Any of the GPIO pins can be designated in software as an input or output pin and used for a wide range of purposes. Two 5V pins and two 3V3 pins are present on the board, as well as a number of ground pins (0V), which are not configurable. A GPIO pin designated as an output pin can be set to high (3V3) or low (0V). A GPIO pin designated as an input pin can be read as high (3V3) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software [7]. Figure 2.3 shows GPIO pinout of Raspberry Pi 3 B+

2.3.3 DHT22 Temperature and Humidity Sensor

DHT22 is a temperature and humidity sensor with 3 to 5V power and I/O. It uses 2.5 mA max current during conversion while requesting data. DHT22 is good for 0-100 % humidity readings with 2-5 % accuracy and good for -40 to 80 °C temperature readings 0.5 °C accuracy. It has a sampling rate of 0.5 Hz. Figure 2.4 shows a DHT22 sensor.

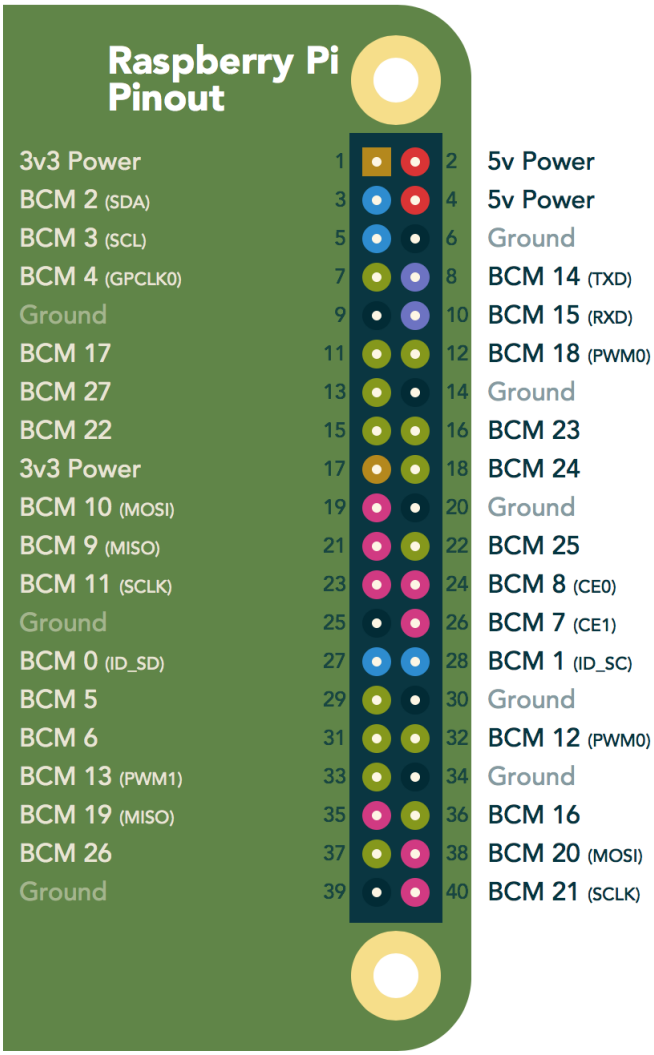

Figure 2.3: Raspberry Pi 3 B+ GPIO Port

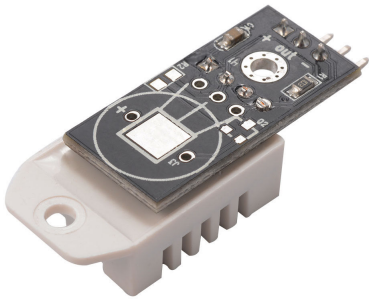

Figure 2.4: DHT22 Temperature and Humidity Sensor

2.3.4 Load Cell and HX711

A load cell is a transducer that is used to create an electrical signal whose magnitude is directly proportional to the force being measured. A load cell usually consists of four strain gauges in a Wheatstone Bridge configuration. The gauges themselves are bonded onto a beam or structural member that deforms when weight is applied. In most cases, four strain gauges are used to obtain maximum sensitivity and temperature compensation. HX711 is a precision 24-bit analog to-digital converter (ADC) designed for weigh scales and industrial control applications to interface directly with a bridge sensor. The input multiplexer selects either Channel A or B differential input to the low-noise programmable gain amplifier (PGA). Channel A can be programmed with a gain of 128 or 64, corresponding to a full-scale differential input voltage of 20mV or 40mV respectively, when a 5V supply is connected to AVDD analog power supply pin. Channel B has a fixed gain of 32. The HX711 load cell amplifier is used to get measurable data out from a load cell and strain gauge. Diagram of HX711 Amplifier is shown in Figure 2.5

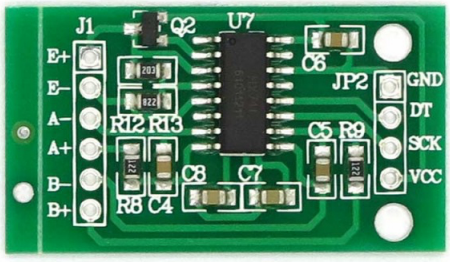

Figure 2.5: HX711 ADC Load Cell Amplifier

2.3.5 Node.js

Node is an asynchronous event driven JavaScript runtime, which is designed to build scalable network applications. Node is similar in design to, and influenced by, systems like Rubys Event Machine. Node takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems there is always a blocking call to start the event-loop. HTTP is a first class citizen in Node, designed with streaming and low latency in mind. This makes Node well suited for the foundation of a web library or framework.

Node.js shines in real-time web applications employing push technology over web sockets [8].

2.3.6 MongoDB

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemata. MongoDB is developed by MongoDB Inc., and is published under a combination of the Server Side Public License and the Apache License. MongoDB supports field, range query, and regular expression searches. Queries can return specific fields of documents and also include user-defined JavaScript functions. Queries can also be configured to return a random sample of results of a given size. MongoDB provides high availability with replica sets. A replica set consists of two or more copies of the data. Each replica set member may act in the role of primary or secondary replica at any time. All writes and reads are done on the primary replica by default. Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically conducts an election process to determine which secondary should become the primary [9].

2.3.7 Socket.IO

Socket.IO is a JavaScript library for realtime web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven. Socket.IO primarily uses the WebSocket protocol with polling as a fallback option, while providing the same interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O [10].

2.3.8 Python

Python is an interpreted high level programming language for general purpose programming. Python features a dynamic type system and automatic memory management. It supports multiple programming paradigm, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library. Python provides good support for programming the Raspberry Pi and provides well documented libraries for interfacing with various sensors connected through the GPIO ports [11].

2.3.9 Swift

Swift is a powerful and intuitive programming language for macOS, iOS. From its earliest conception, Swift was built to be fast. Using the incredibly high-performance LLVM compiler, Swift code is transformed into optimized native code that gets the most out of modern hardware. Swift is the result of the latest research on programming languages, combined with decades of experience building Apple platforms. Named parameters brought forward from Objective-C are expressed in a clean syntax that makes APIs in Swift even easier to read and maintain. Some of the new features of are closures, tuples, generics, structs that support methods, extensions and protocols, functional programming patterns etc. [12]

2.4 Related Work

Many studies have been done in this sector in last few years. Most of the studies only tried to monitor the temperature or check other internal environment factors. Very few works are done to establish an active food inventory monitoring system. Now we will explore some of the works done previously. We will also try to figure out strength, opportunities and limitation of their work.

2.4.1 A Food Monitoring System Based on Bluetooth Low Energy and Internet of Things

In 2017 Mr. A. Venkatesh et al. [13] tried to the make a food monitoring system using temperature and VOC sensor. They identified that many food poisoning diseases take place due to improper food refrigeration. So they housed a VOC gas sensor for monitoring the food condition. But due to complexity of biological olfaction, the artificial olfactory system has non linearity characteristics. So it is hard to detect accurate food condition using only one type of VOC sensor. Their work also features a cloud database and web app for alerting the users. An illustration of the block diagram of their system is shown in figure 2.6

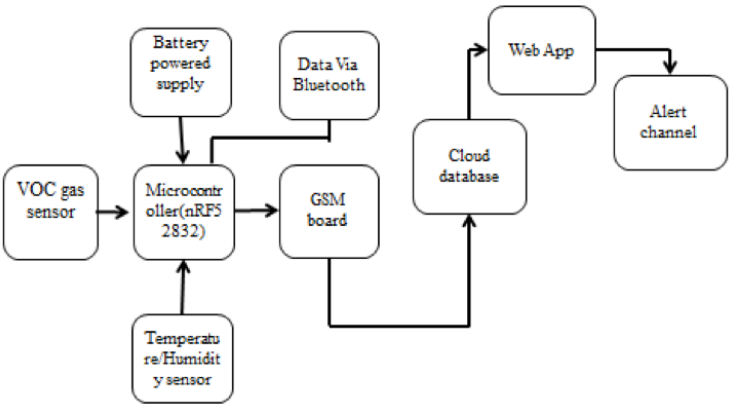

Figure 2.6: Block Diagram for Food Monitoring System Based on Bluetooth Low Energy and Internet of Things

2.4.2 iTrack: IoT Framework for Smart Food Monitoring System

In 2016 Srivastava et al. [14] tried to make a food monitoring system which houses temperature sensor, humidity sensor, light sensor, magnetic sensor, ultrasonic sensor, air quality sensor. All the sensors are used to monitor different factors of the food. Due to more sensors food quality evaluation is more accurate. But the project doesnt contain any support for remote food monitoring. There is no active database to store the readings so that users can draw to a point. Figure 2.7 illustrates the proposed solution for iTrack framework for food monitoring system.

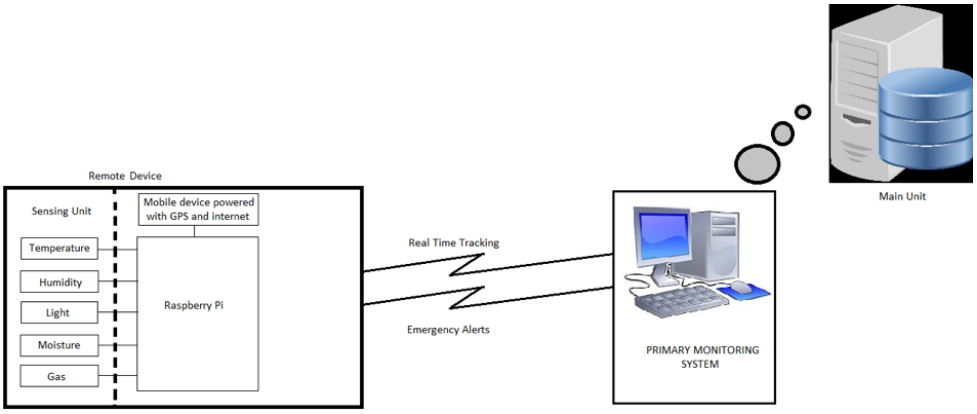

Figure 2.7: Proposed iTrack Solution

2.4.3 RFID Based Smart Fridge

A. Hachani, et al. made an attempt to design an automated IoT based smart fridge [15]. They proposed a pilot design of a RFID based smart fridge. The fridge can detect what kind of food is inside via sensing the RFID tags labeled with each food. Then the fridge can monitor every thing and manage accordingly. Embedding RFID provides an elegant solution to such a problem of assessing whats inside in the fridge. RFID tags can uniquely identify each food inside to refrigerator and tag them in the system. But the fact is that few if any food items have RFID tags. Thats why that design fails to achieve its goal. The design doesnt provide feature to turn on/off remotely.

2.4.4 Design of Smart RFID Tag System for Food Poisoning Index Monitoring

In 2011 Chang Won Lee et al. [16] proposed a Smart RFID tag which is a package of different sensors. Proposed system measures the temperature and humidity using the smart RFID tag. The measured information can be calculated according to food poisoning index with four grade, interest, caution, warning, risk. The sensor readings are then evaluated through a food poisoning index. The proposed system confirms usefulness through experiments. These systems can be advantageous if it is integrated with IoT technology.

2.4.5 Samsung Family Hub Refrigerator

Samsung Electronics have developed the Samsung Family Hub Refrigerator [17] which is one of the most noteworthy IoT smart refrigerator. It has vast functionality like large LCD display, internet connectivity, media access. The refrigerator even comes with a camera to monitor what food items are there at a moment. But the system didnt come with any automated food management mechanism to assess the quantity of food. The refrigerator doesnt have any measures for active food inventory monitoring. So users cant get notified when there is shortage of food.

2.4.6 Design and Implementation of Food Monitoring System Based on WSN

Kong Xiangsheng et al. [18] made an attempt to design a remote wireless monitoring system for food supply network based on Zigbee and RFID which are mainly used to detect and gather food supply information and upload information to monitoring center. They approached a low-complexity, low-cost, low-data-rate and low-power-consumption design principles and food data clustering approach for Wireless Sensor Networks(WSN). Main disadvantage of the the system is it can't work in the refrigerator and can't be a all around solution for monitoring food in the refrigerators. User have to use it manually to monitor desired food.

2.4.7 A Survey on Monitoring and Control system for Food Storage using IoT

In 2017 Rohan et al. [19] proposed a Smart food monitoring unit which governs control over various parameters causing decay or rotting of food materials, therefore ensuring appropriate quality of food during various atmospheric changes. The is implemented as a whole system for the user to monitor the foods with metrics such as temperature and weight. Main disadvantage is being a whole system it can not be implemented on any other refrigerator or any cooling unit other then its own cooling solution and also unable to provide visual feedback to the user.

Chapter 3

System Architecture and Design

3.1 Architecture of the Smart Food Monitoring System

The proposed Smart Food Monitoring System consists of five components. They are real time food inventory monitoring, real time temperature monitoring, temperature logging, visual feedback and user application. Logical Architecture of the entire system is shown in Figure 3.1. The system consists of a sensor unit where Raspberry Pi, all the sensors such as temperature sensor, load cell, camera are housed. The whole system works on an ecosystem of these three module. The user application requests actions, the server processes the action and the instruct the Pi to approach for the next step. All the logical works are done in server side of the system. The user application acts as an action input and result output module, the server acts as the data and request processor unit and the sensor unit works as the data input or collector unit. When monitoring configurations are turned on, all the sensors, except the camera constantly gets reading from the environment and processed them. The processed raw data is then sent to the server.

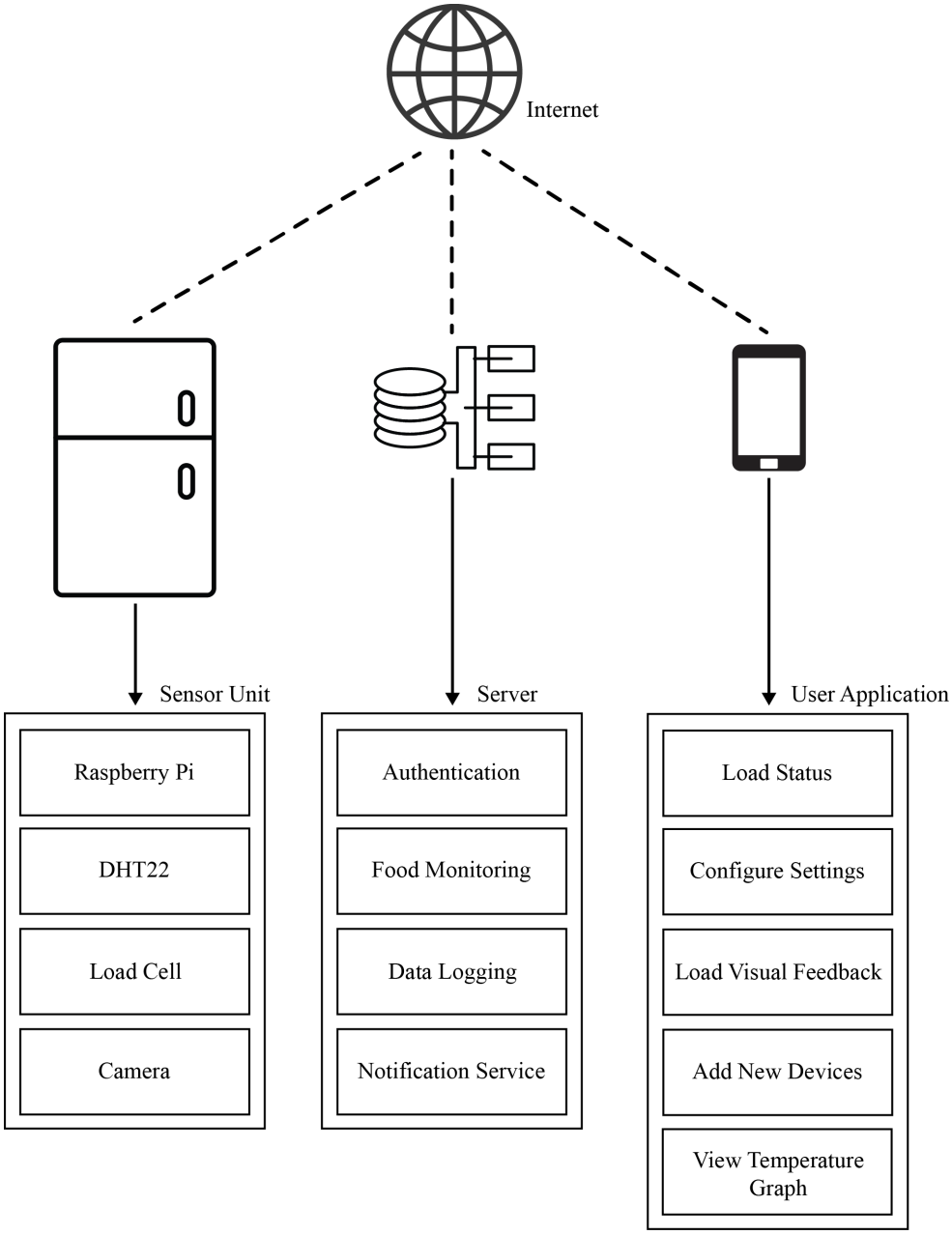

Figure 3.1: System Architecture of the Smart Food Monitoring System

3.1.1 Server Architecture

The Server is the center of all the procedure. Both the user application and the device application communicates with each other via the server. The server manages all the requests and stores data according to the monitoring logic. Server application is written using Node.js JavaScript framework. Server handles all the event actions and responses to the requests via Express library. All requests reaching the server are processed asynchronously which reduces the queuing time

for response. Server Server application listens to various POST, GET, PATCH, DELETE requests via Express [20]. These are used to query for various device informations from the server. On the other hand for gathering real-time output from user to device or vice versa the server uses Socket based communication which is made possible using Socket.IO library. The Server uses NoSQL database which can store any sort of arbitrary type of data without any table. It is necessary because the database requires to constantly store temperature data. Server Architecture is illustrated in the figure 3.3. Main server process is distributed into 5 components. The Main Thread is divided into request handler and listener. Request handler in subdivided into HTTP request handler and Socket event handler. The Database handler maintains connection between the server and the database. The Database Model is divided into Device Schema and User Schema. There is a middleware for authentication which takes place on almost all of the user request. Then there is the configurator file which configures which port to listen.

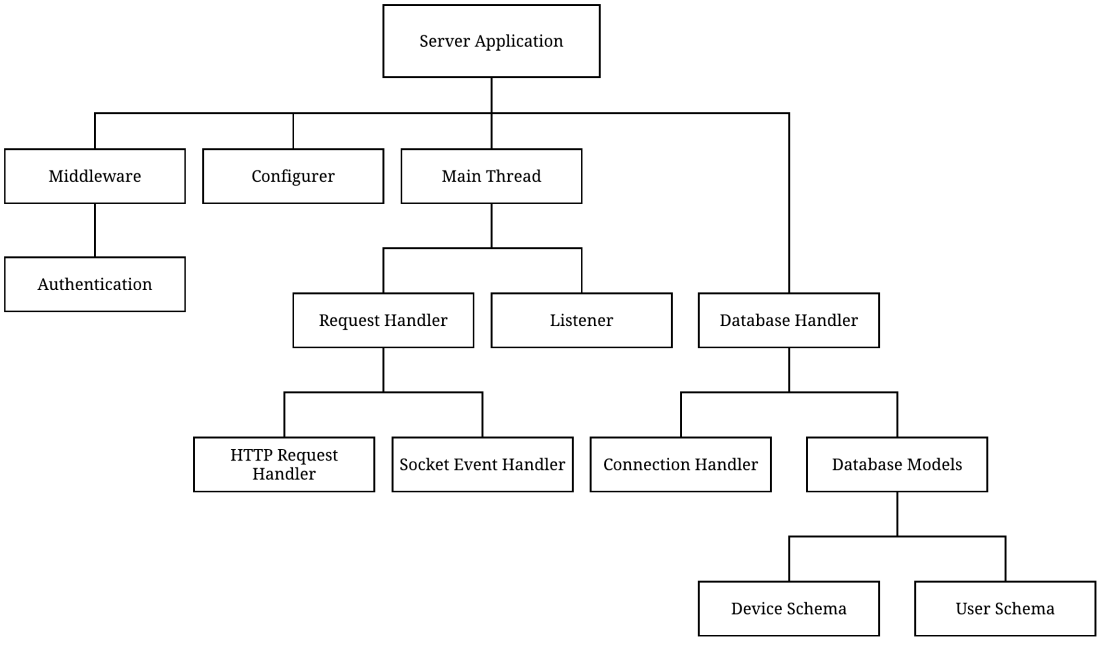

Figure 3.2: Server Architecture

3.1.2 User Application Architecture

The user application is the medium for the user to interface with the whole system. The user has to be connected to the server through network connection so that the application can communicate with the device via server and query various information from the server. The user can check device status and current sensor readings via the application. The user application can be used to configure the settings regarding the way the server responses to temperature and sensor warnings. These parameters are then saved in the database of the device schema which is checked every time the device provides a sensor entry. If the sensor reading does not meet the given conditions of the user then the user is notified from the server. These notifications are actually achieved through the application notification handler, which queries for notification in every six seconds. The server flags the sensor readings if the conditions are not met. If the application notifier queries and gets a flagged reading then it triggers a notification in the users device. The user can take actions according to the notification event. Figure 3.4 demonstrates the user application architecture.

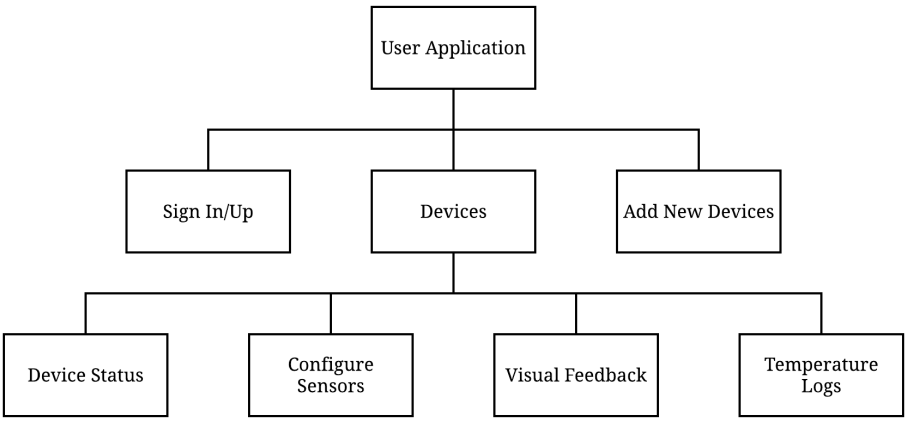

Figure 3.3: User Application Architecture

3.1.3 Database Design

Database design is always an important part of database based system. A poorly designed database can cause failure of the entire system. Smart Food Monitoring System requires a database for device information and user information. Hence we have to different models. One for the user and the other for devices. Our

system has two entity. The device and the user. The Device entity has attributes like device id, serial number, configurations, current temperature, current weight, logs. On the other hand the User has user id, user email, password, devices, tokens. The Device entity and the user entity has a relationship named Monitors. User and Device entities have many to many relationships. Figure 3.5 illustrates the Entity Relationship diagram of the Smart Food Monitoring System (SFMS).

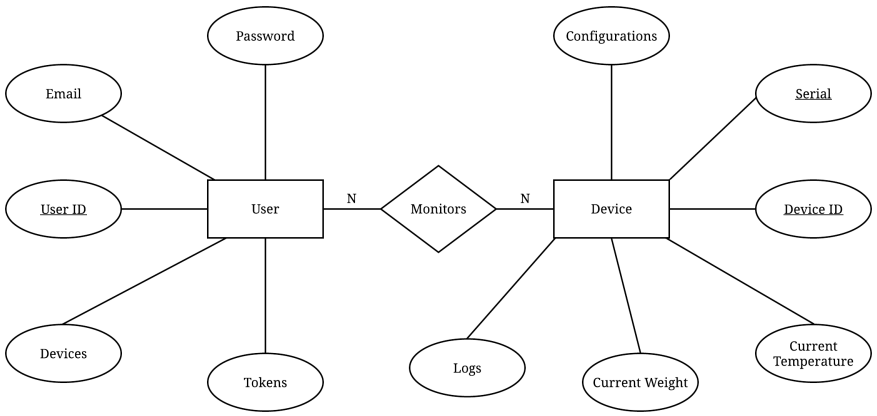

Figure 3.4: ER Diagram of SFMS

3.1.4 Sensor Unit Workflow

The Sensor Unit does not evaluate any sort of sensor data. All sort of processed data received from the sensors are pushed to the server where the data is further processed according to the monitoring logic and if appropriate the data is then logged into the database for later inspection. The heart of the sensor unit is the Raspberry Pi [21] which executes all the sensor reading according to schedule. All data transmission of the Raspberry Pi is done using the servers REST api and Socket.IO client events. These methods enable the sensor unit to transmit sensor data in continuous manner. Both DHT22 and Load Cell needs a cool down time before reading consecutive data. That's why between each reading there is a six seconds gap. Every six seconds the Raspberry Pi inquiries for environmental readings if the monitoring setting is turned on. While listening for any user request via server if there is a visual feedback request than the Pi uses the fswebcam library to capture image using the camera and upon successful image capture the data is encoded in base64 data and pushed to the server to

send back to the user application. Figure 3.2 illustrates the working procedure of the sensor unit.

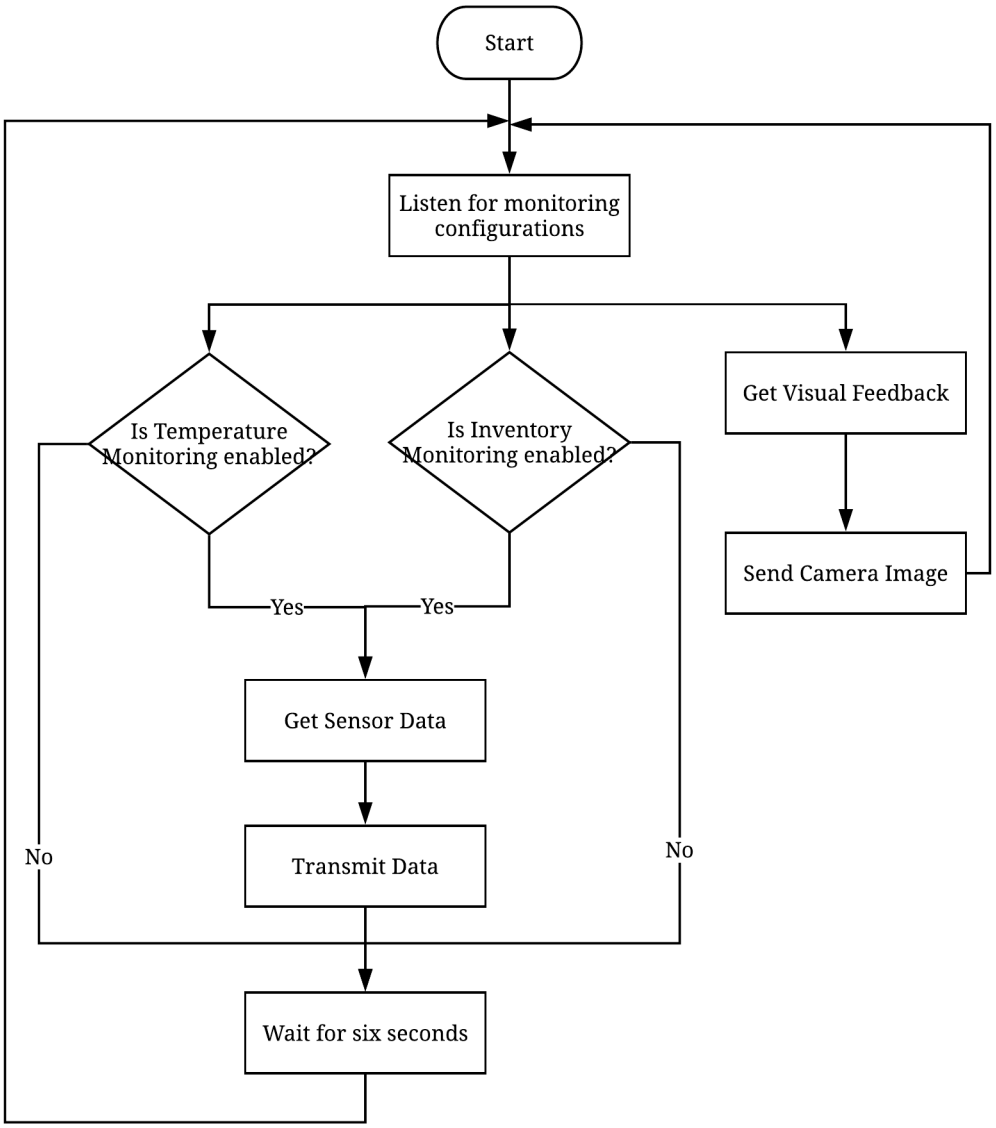

Figure 3.5: Workflow of the Sensor unit

Chapter 4

System Implementation

The entire system implementation consists of hardware for sensor reading, device application for automating the sensor reading procedure, user application for users to monitor and configure the system. The sensors are connected with Sensor unit via serial cables. Device application sends the information to server over Wi-Fi. The user application connects to the server over the local area network.

4.1 Implementation Tools

For deploying this project in software and hardware part, we need prerequisite to be maintained. These are of three categories:

- Devices and Instruments Requirements
  - Personal Computer (2.8 GHz Processor, 16 GB RAM)
  - Raspberry Pi 3B+
  - DHT22 Temperature Sensor
  - HX711 ADC Amplifier with Load Cell
  - Network Connection with Server PC
  - USB Camera
  - iOS Device for Client User
  - Wi-Fi Router
  - Power Adapter

- Software Requirements
  - Visual Studio Code
  - Xcode
  - Terminal
  - OS for Server Application Windows 7/8/8.1/10, MacOS, Linux (32/64 bit)
  - For Client User Application minimum iOS 9
- Programming Language and Framework
  - Node JS v10.9.0
  - Python v2.7.15
  - Socket.IO
  - MongoDB
  - JavaScript
  - Swift 3

4.2 Circuit Connection

Circuit Diagram for Raspberry Pi and all the sensors is shown in Figure 4.1. For all pin connections BCM pin numbering [8] convention was used. Here we have connected DHT22 temperature and humidity sensors VCC pin to a 3v3 pin, DT pin to GPIO 17, GND pin to a ground pin. We used an HX711 ADC load cell amplifier to read the load cell outputs. HX711 has 4 pins connected to the Pi and 4 pins to the Load Cell. The DT pin of the amplifier is connected to the GPIO 9 pin and the SCK pin is connected to the GPIO 11 pin of the Raspberry Pi. Remaining two pins are connected to any 3v3 VCC and ground. On the other side Load cell red wire is connected to the E+ pin, black wire is connected to the E- pin, white white is connected to the A+ pin and green wire is connected to the A- pin of the amplifier. It should be noted that the data connection of the HX711 is highly prone to outside noise which can lead to undesirable readings. Thats why it is required to solder the connections of the HX711 and the Load

Cell before reading data. All circuits are connected using the male to female wire connecters.

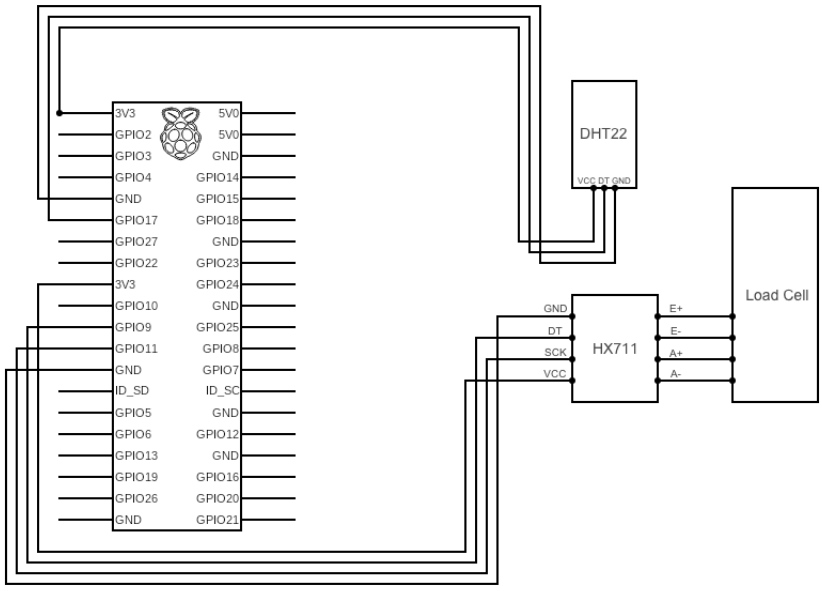

Figure 4.1: Sensor Unit circuit consisting Raspberry Pi and Sensors

4.3 Peripheral Connection

There is only one peripheral connection with the Raspberry Pi. We used the built in USB 3.0 ports of the Pi to connect with the camera which is used to capture still images of the inventories. Figure 4.2 shows the peripheral connection and the power source connections of the Raspberry Pi.

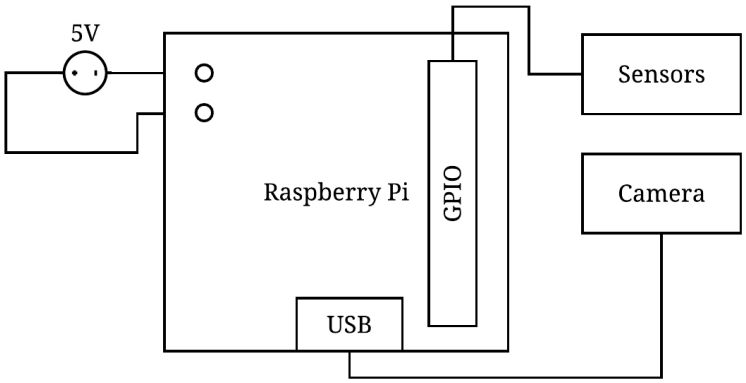

Figure 4.2: Peripheral Connection with the Raspberry Pi

4.4 Application User Interface

Smart Food Monitoring System provides a mobile based user application. This application enable users to know some vital information about the their refrigerators. Users can check the current fridge temperature, current weight of inventory contents, get notified if the inventory is below certain threshold and if temperature is above specified temperature. The user application provides simple and intuitive controls which can helps to use the application with ease and without missing any feature. Figure 4.3 and 4.4 shows the Graphical User Interface of the User Application.

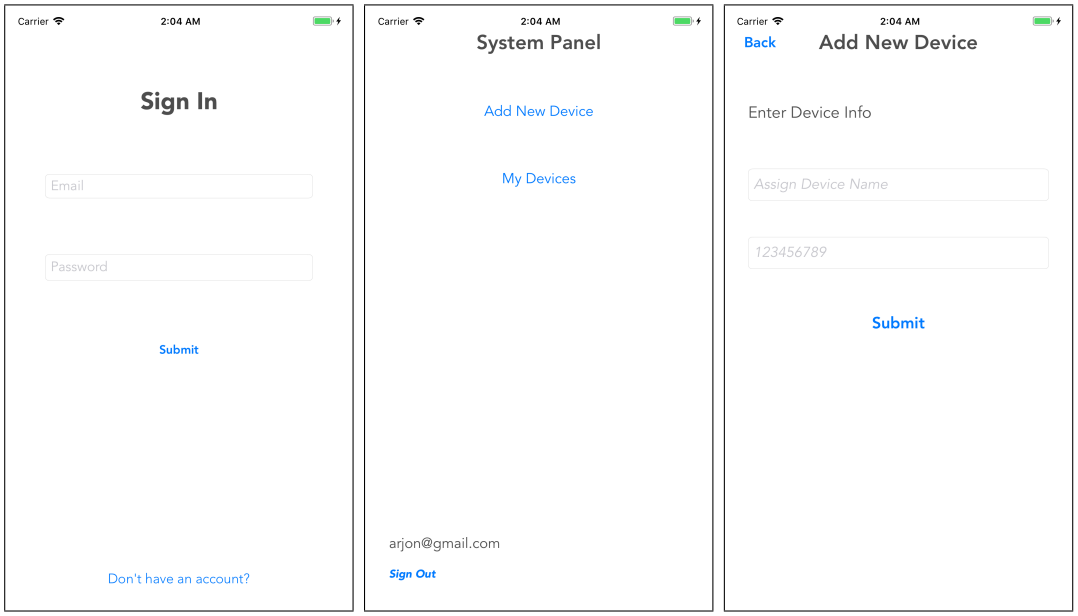

Figure 4.3: User Application Interface 1

From both figures we can observe some of the application features provided to the user. The application also provides server generated notifications which are presented when application is in both active or idle state.

4.5 Experiments

We carried out some experiments with our proposed system to evaluate the performance and throughput of the system. For accurate working evaluation our experiments were done using a commercial refrigerator and most of the scenarios were simulated to mimic real life situations.

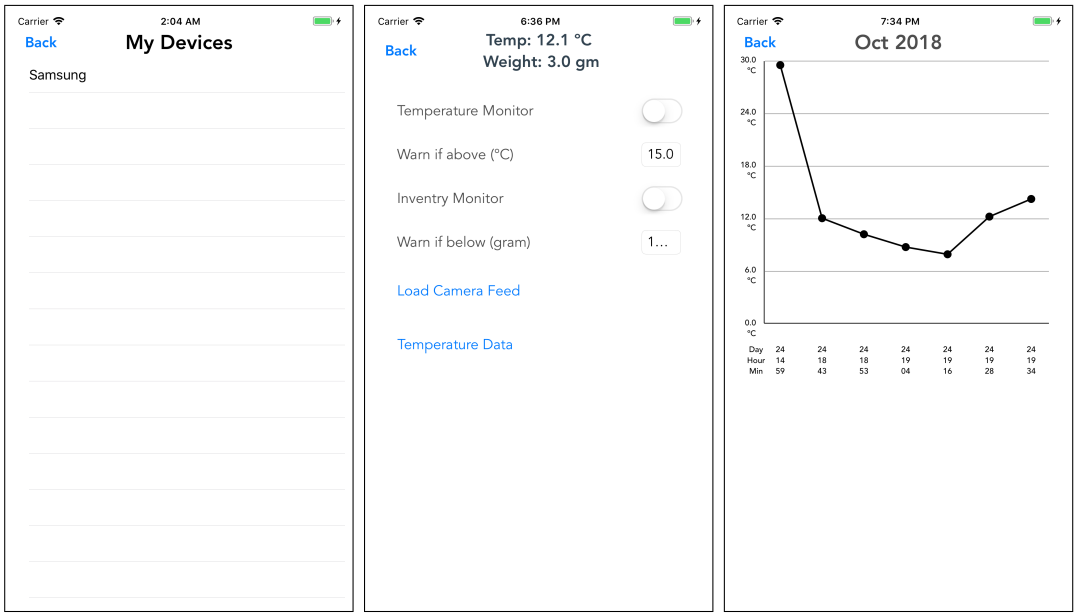

Figure 4.4: User Application Interface 2

4.5.1 Experiment Environment

The experiment was carried out using the office refrigerator of the Department of Computer Science and Engineering, CUET. The refrigerator environment temperature was fluctuating around 13°to 14°C. On software side of things virtual environment for python was used to run third party packages on the Raspberry Pi. The experiment was captured using a camera phone for video reference.

4.5.2 Experimental Setup

For successful experimentation it is required to situate sensors inside the refrigerator. The Smart Food Monitoring System sensors were attached to side rails inside the refrigerator. The Load Sensor was setup on a metal platform on which we can put any object to measure the weight of the object. The whole load cell setup was housed on one of the refrigerator trays. The camera was situated on a convenient place inside the refrigerator so that is can capture quite a view of the whole tray. Due to condensation property all electronics were handled with care when operated inside a refrigerator. It was best not to situate the sensors inside the freezer where temperatures are freezing. Figure 4.5 shows the setup of temperature sensor and camera inside the fridge and figure 4.6 shows the weight sensor setup.

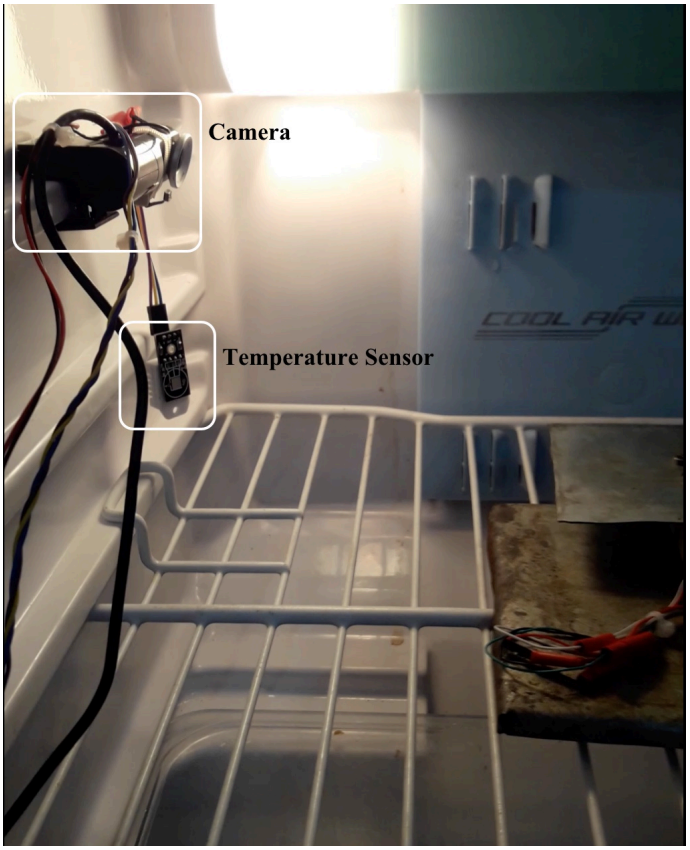

Figure 4.5: Camera and Temperature sensor setup

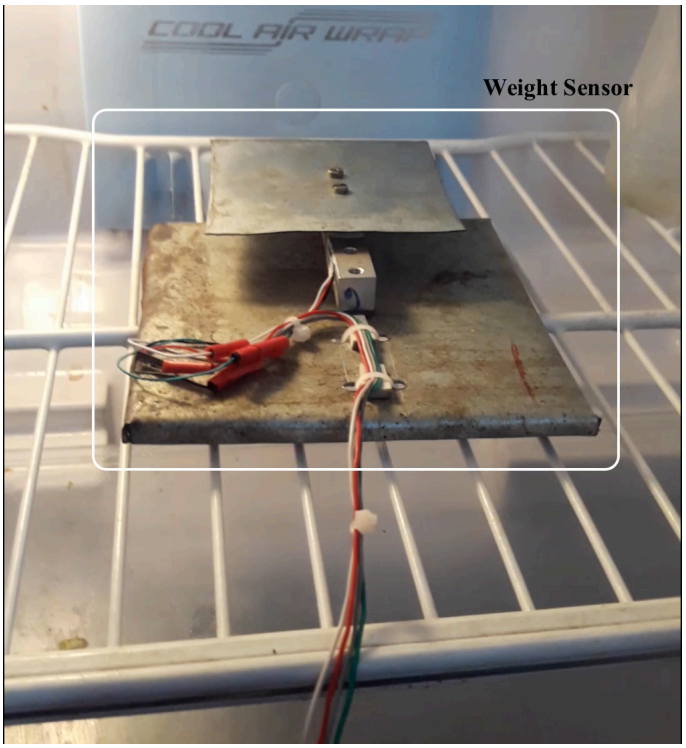

Figure 4.6: Weight Sensor Setup

The Server application which controls the whole system was running on a MacBook Pro with Core i7 processor and 16 GB of RAM. The server wasnt hosted on the internet. Hence localhost address was used to establish server connection between the Raspberry Pi and the User Mobile Device. For stability NPM package manager was used to maintain the packages used in the server application. For Raspberry Pi we used PIP package manager for python. The Device Application was run on a Quad Core 64 bit processor at 1.4 GHz, consisting 1 GB of RAM and a built in Wi-Fi modem. The experimental setup allowed us to collect the following information:

- Real Time Temperature Monitoring
- Real Time Inventory Monitoring
- Visual Feedback of Inventory
- Graphical Representation of logged temperature data
- Simulation of inventory alert
- Simulation of temperature alert

4.5.3 Experiment Procedure

To carry out the experiment first we created a user account providing a email address and password. Then we added a new device to be monitored from the user application. To add a device it need to enter the serial number assigned to it. The we provided a name to the device. When the device is up and running we opened the device control panel by selecting it from 'My Devices' option. There we enabled the 'Temperature Monitor' and 'Inventory Monitor' option. Initially the device read a temperature of 13.6°C and weight of 0 grams. Then we set 'Warn if above (°C)' option to 4.0 and 'Warn if below(gram)' option to 250. Then we put a 500ml bottle filled with water on the weight sensor tray. We observed how the system behaves in these situation and then removed the bottle to see how it behaves in response. Figure 4.7 shows the initial setup parameters on the user application.

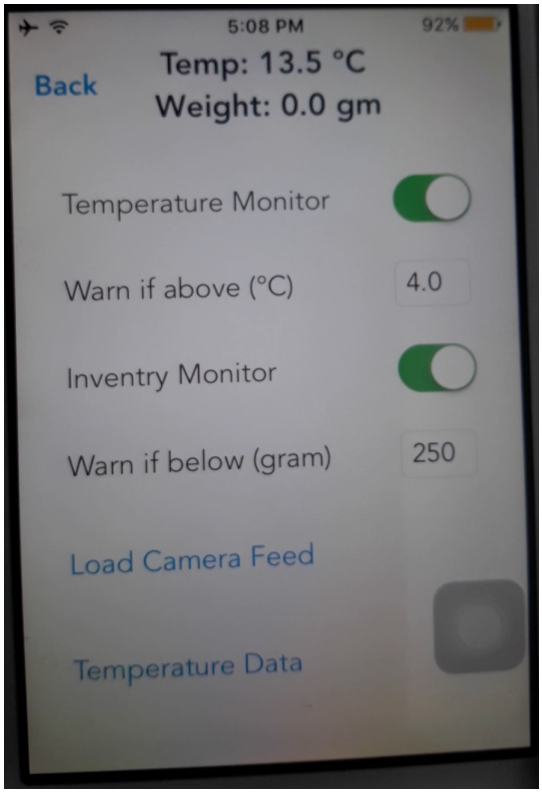

Figure 4.7: Initial Setup parameters on User Application

4.5.4 Evaluation Methods

We have performed several tests in different place and environment. While testing we have measured correctness, accuracy and performance. In this chapter we will represent evaluation of our system. We can evaluate our system in the following three methods:

Qualitative Evaluation

Qualitative methods is simply valuation using non-numerical data. It has advantage that the researcher is able to dig deep into the issue. For example, if a statics show that X percent of people will vote for party A. This statics is quantitative information. It can answer the query that how much people will vote for A but can't answer the query that why those people will vote for A. Qualitative methods try to find the answer of second query. However, qualitative evaluation in a big sample is very difficult.

Quantitative Evaluation

Quantitative evaluation is simply valuation using numeric data. Evaluation are made using scientific tools and measurement. This method is suitable for large sample space.

Subjective Evaluation

Subjective evaluation is an assessment or evaluation that is highly influenced by person's feelings. It is based on participant feedback. User of subjective evaluation can give a better overall picture of performance. However, subjective evaluation has risk of biasing.

We will use quantitative evaluation method to evaluate our system.

4.5.5 Evaluation Measures

We will evaluate the system accuracy and performance according to its temperature reading accuracy, weight sensing accuracy, notification responsiveness.

Temperature Reading Comparison

Evaluating the systems temperature sensing accuracy to check if for any given moment the temperature sensor reading is actually authentic and good reading by comparing its reading with a already established sensor.

Weight Reading Comparison

Evaluating the systems weight sensing accuracy to check if for any given moment the weight sensor reading is actually valid by comparing its reading with a weighting scale.

Notification Responsiveness

Evaluating the systems responsiveness according to the awareness of the system by surveying the time taken by the system to response to the event of removing weights below to the warning limit.

4.5.6 Results

In this section the review of the system carried out through the experimental data will be discussed. We will also discuss accuracy of some of our system sensors and stat how reliable it can be in case of triggering notification events.

Temperature Reading Comparison

From table 4.1 we can observe the temperature read by commercial thermometer and our system. These readings were taken over time with arbitrary time intervals. We can also see the accuracy of the temperature sensor reading of our system opposed to the fridge reading from the commercial thermometer which is crucial for authentic result. From figure 4.8 we can observe the difference of temperature readings between our systems temperature sensor and an established temperature sensor. From the graph we can observe that the temperature reading difference between the two sensors are not too apparent. Our system provides a quite accurate temperature data hence it is reliable for the system for monitoring with correct measurement of temperature. From 4.1 table we also get a calculated average temperature reading accuracy of 96.8%.

Table 4.1: Temperature readings from commercial thermometer and our system

Time	Fridge Reading (ºC)	System Reading (ºC)	Accuracy (%)
8:54 PM	4.5	4.3	95.5
9:31 PM	4.9	4.7	95.9
10:04 PM	5.2	5.4	96.1
10:31 PM	5.4	5.5	98.1
11:01 PM	5.5	5.7	96.3
11:12 PM	4.5	4.2	93.3
11:41 PM	4.9	4.6	93.8
12:11 AM	5.1	4.9	96.0
12:32 AM	5.3	5.4	98.1
1:13 AM	6.3	6.2	98.4
1:41 AM	6.3	6.3	100
2:06 AM	6.8	6.8	100

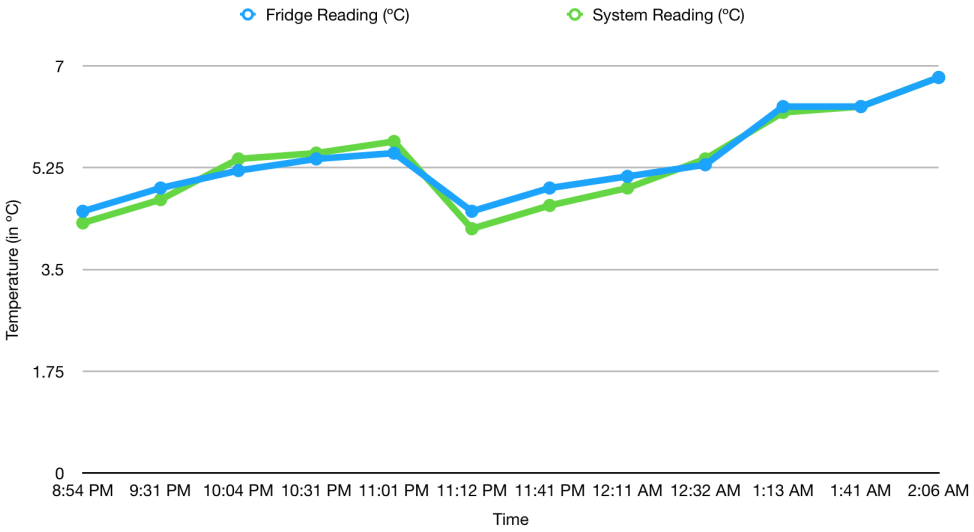

Figure 4.8: Comparison of Temperature readings over time

Weight Reading Comparison

From table 4.2 we can observe the weight amount read by commercial weighting scale and our system taken over ten trials. These readings were taken with different arbitrary weights and read on both our system and commercial solution. The accuracy of the weight sensor reading of our system opposed to the the commercial solution is used to calculate average weight reading accuracy of 97.6%.

Table 4.2: Weight readings from commercial weighting scale and our system

Number of Trials	Weighting Scale Reading (gm)	System Reading (gm)	Accuracy (%)
1	10	10	100
2	12	11	91.6
3	15	15	100
4	18	17	94.4
5	20	21	95.0
6	24	24	100
7	30	30	100
8	35	34	97.1
9	40	40	100
10	50	49	98.0

From figure 4.9 we can observe the difference of weight readings between our systems weight sensor and a weighting scale. From the figure we can observe that the weight scale measurements and the systems load cell measurements are quiet

close. So our system provides a quite accurate weight data hence it is reliable for the system for monitoring with correct measurement of weight and notify the user.

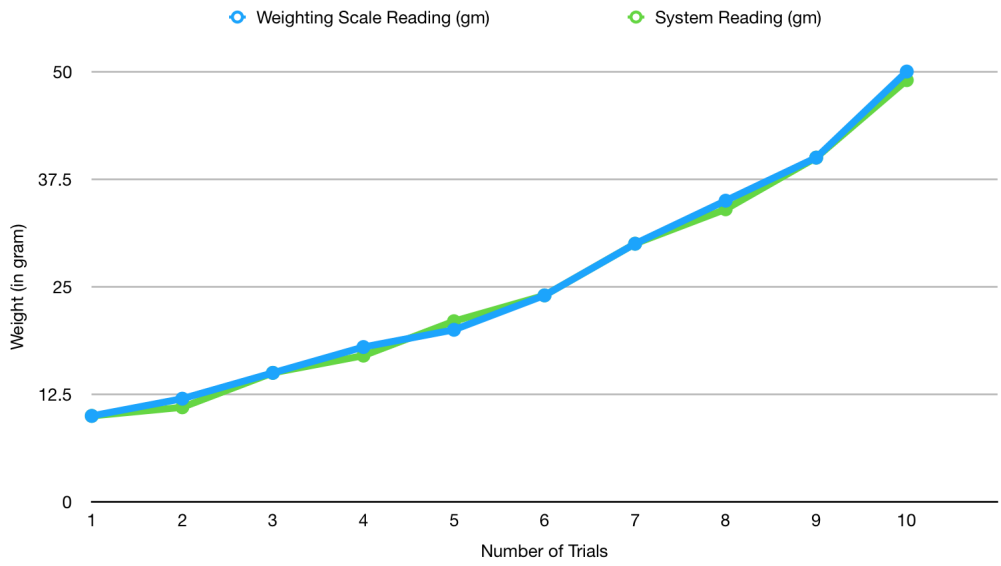

Figure 4.9: Comparison of Weight readings over time

Notification Responsiveness

In table 4.3 response time for different notification responses are situated. From these table we can calculate the average system response time. Calculating the average response time we get a value of 35.8 seconds. So it can be stated that our system can notify the user about inventory lacking in average of 36 seconds.

Table 4.3: Notification response time of the system

Number of Trials	Notification Response Time (in second)
1	31
2	35
3	37
4	33
5	39
6	38
7	31
8	42
9	39
10	33

From figure 4.10 we can survey the time taken by the system to notify or warn the user about inventory warnings. As the server logic looks for five consecutive below threshold data and each readings takes place on the sensor unit exactly in a interval of six seconds hence there is always a delay of thirty seconds present on the system.

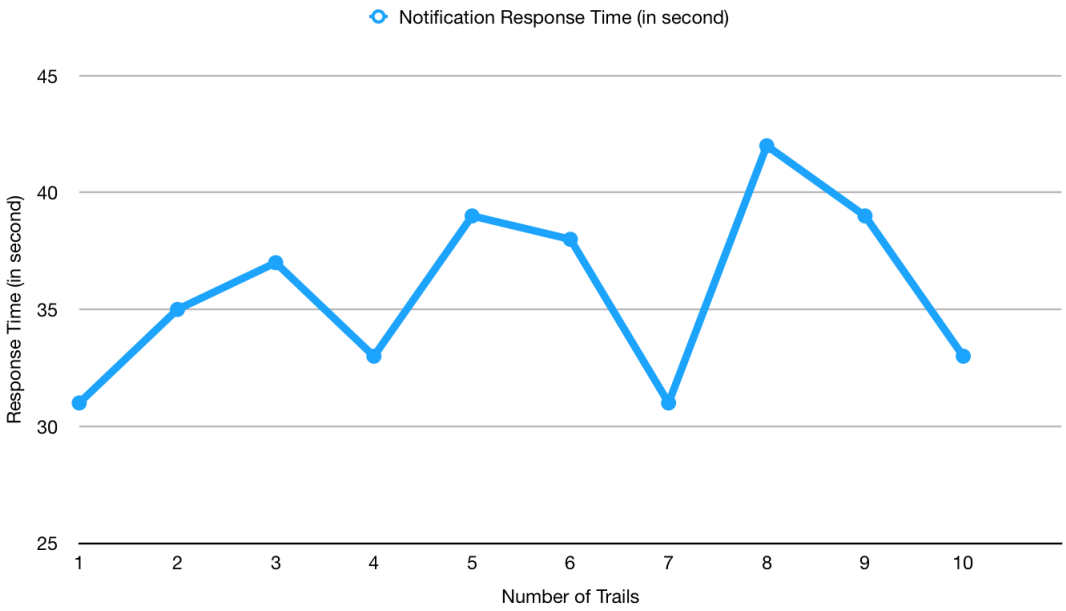

Figure 4.10: Notification response time of the system

Application Feature Outputs

To check the refrigerator inventory the option "Load Camera Feedback" can be selected. This provides the instant visual image of the inside storage of the refrigerator. Figure 4.11 shows the camera feedback option in action. In figure 4.12 we see the notifications sent from the server warning user about temperature and inventory. And in figure 4.13 we see the application representing graph presentation of temperature reading history.

Figure 4.11: Load Camera Feedback option providing image of refrigerator inventory

Figure 4.12: Notification sent from the server to user about temperature and inventory warning

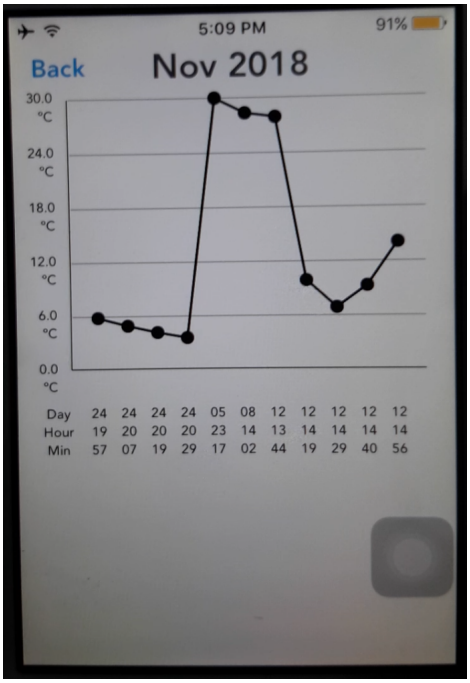

Figure 4.13: Temperature Data option providing temperature history on graph representation

4.6 Discussion

The main objective of this project is to develop a system that will help users to monitor their daily food content inside the refrigerator and provide them an additional control over their grocery. Aside from these the another objective of the project is to reduce food wastage through the developed system. To meet this goal, we have developed a prototype of food monitoring sensor unit, a real time monitoring application for the users. Our prototype of Smart Food Monitoring System was able to monitor food for a specific compartment of the refrigerator. Although our prototype is not compatible of monitoring the whole fridge but an extensive version of the system can be developed using the same procedures to monitor the whole fridge.

Chapter 5

Conclusion

We have worked with real time IoT based technology with food monitoring for refrigerators. Food monitoring is very important as it makes life bit easier by giving access to the whereabouts go the food inventory of the refrigerator and helps to stop wastage of food due to prolong storage of food. In this chapter we will present a overview of our system with its limitation and future recommendation.

5.1 Conclusion

The main objective of this project was to develop an IoT based system through which user can monitor the food contents inside their refrigerator. To meet the goal we developed a server that can manage and store data and response to users according to their request. Then we developed a application for the sensor unit device which can read environmental data through the transducers and process them to human recognizable data. The data is then sent to the server where it is stored and processed further for logical implementation. We also developed an User application which can be used to monitor and get status of the refrigerator in real time and configure when to notify in case inventory runs low or refrigerator temperature gets high or get to see whats inside the fridge. We successfully met most of our objectives like helping the user to decide for grocery, prevent from access food stocking, reviewing whats inside the fridge or getting squinted if the fridge is performing at its optimal performance. Users can reliably use our system to monitor certain metrics. But there is also limitations. Our system cant provide a full refrigerator monitoring solution as it is very tricky without the integration

of the system with the fridge that can only be possible by the manufacturer of the fridge. As load cells provide unstable readings, which sometimes generates values which are not desirable for the system procedure. It is also disadvantageous for one camera to get a extensive look of the whole fridge from inside. Due to condensation some sensor might not work the way they are intended to.

5.2 Future Recommendations

Very few work has been done previously in these sector. Hence there is a lot of room for improvements. We have only worked in these project some of important monitoring criteria. We also addressed some of the prominent problems. We also tried to make our system simple as possible. Thats why we have rejected some interesting ideas like using image detection, poisonous gas monitoring, RFID tagging etc. So there are a lot of scope to work in the future. The future recommendations are:

- Using AI to detect food from the imagery and notify the user whats inside their fridge.
- Integrating RFID tag reader for identifying kind of food is currently inside the fridge.
- Integrating Poisonous gas detecting sensors to evaluate the freshness of the fridge environment.
- Integrating a food suggestion system based on there shelf life.

If we can implement this system in our daily life style, then it is possible to prevent a lot of food wastage and give ourselves some assistance for doing the right grocery and lead to a healthy food consumption.

References

[1] T. N. Kosmatos, E.A. and Boucouvalas, "Integrating rfids and smart objects into a unified internet of things architecture," *Advances in Internet of Things: Scientific Research*, 2011. [Online]. Available: http://dx.doi.org/10.4236/ait.2011.11002

[2] Food and Drug Administration, "Refrigerator Thermometers: Cold Facts about Food Safety." [Online]. Available: https://www.fda.gov/food/resourcesforyou/consumers/ucm253954.htm

[3] R. Aggarwal and L. Das, "Rfid security in the context of internet of things," *First International Conference on Security of Internet of Things*. [Online]. Available: http://dx.doi.org/10.1145/2490428.2490435

[4] S. T. Somayya Madakam, R. Ramaswamy, "Internet of things (iot): A literature review."

[5] World Wide Web Consortium, "Relationship to the world wide web and rest architectures," February 2004. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/relwwwrest

[6] Raspberry Pi Foundation, "Raspberry pi." [Online]. Available: https://www.raspberrypi.org

[7] Pinout.xyz, "The comprehensive gpio pinout guide for the raspberry pi." [Online]. Available: https://pinout.xyz/

[8] Joyent, "Node.js." [Online]. Available: https://nodejs.org/

[9] MongoDB Inc., "Mongodb." [Online]. Available: https://www.mongodb.com/

[10] Socket.IO, "Socketio." [Online]. Available: https://socket.io

[11] Guido van Rossum, "Python, high-level programming language." [Online]. Available: https://www.python.org/

[12] Apple Inc., "Swift programming language." [Online]. Available: https://developer.apple.com/swift/

[13] S. A. M. K. Mr.A.Venkatesh, T.Saravanakumar, "A food monitoring system based on bluetooth low energy and internet of things," *Int. Journal of Engineering Research and Application*.

[14] A. G. Amrita Srivastava, "itrack: Iot framework for smart food monitoring system."

[15] A. Hachani, I. Barouni, Z. B. Said, and L. Amamou, "Rfid based smart fridge," *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2016.

[16] C. W. Lee, N. T. Van, K. K. Jung, J. W. Kim, W. S. Choi, and K.-H. Eom, "The design of smart rfid tag system for food poisoning index monitoring," *The 2nd International Conference on Software Technology*, 2013.

[17] S. E. America, "Samsung family hub smart refrigerator," 2018. [Online]. Available: https://www.samsung.com/us/explore/family-hub-refrigerator/refrigerator/

[18] K. Xiangsheng, "Design and implementation of food monitoring system based on wsn," *Advance Journal of Food Science and Technology*, 2014.

[19] A. K. R. S. Rohan Wagle, Mayur Shah, "A survey on monitoring and control system for food storage using iot."

[20] Express, "Express a javascript framework." [Online]. Available: https://expressjs.com/

[21] V. Kumawat, S. Jain, V. Vashisth, N. Mittal, and B. Jangir, "Design of controlling home appliance remotely using raspberry pi," *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017.

Appendices

Appendix A

Source Code

Listing A.1: server.js

```
require('./config/config');
const {ObjectID} = require('mongodb');
const express = require('express');
const http = require('http');
const socketIO = require('socket.io');
var bodyParser = require('body-parser');
const _ = require('lodash');
var {mongoose} = require('./db/mongoose');
var {Device} = require('./models/device');
var {User} = require('./models/user');
var {authenticate, superAuthenticate} = require('./
   middleware/authenticate');
var app = express();
var server = http.Server(app);
var io = socketIO(server);
const port = process.env.PORT || 4000;
app.use(bodyParser.json());

//// Device pings every 5 seconds
//// redFlag is number of value warning
const minTimeIntervalBetweenNotifications = 30
```

```
                    /// in seconds
io.on('connection', (socket) => {
console.log('connected to client');

socket.on('join', (room, callback) => {
socket.join(room);
console.log('socket joined',room);
callback(`joined service--${room}`);
var redFlag = 0;
var redInvFlag = 0;
var lastTriggerTime = 0;
var lastInvTriggerTime = 0;
socket.on('logTemp', (tempData, callback) => {
console.log(tempData.data)
Device.logCurrentTemp(tempData).then((device) => {
// console.log(device);
}).catch(err => {
console.log(err);
})
Device.surveyTemperatureData(tempData).then((flag) => {
if(flag) {
redFlag++;
} else {
redFlag = 0;
Device.getDeviceID(tempData).then((device) => {
        User.logWarning(device._id, false, "temperature")
}).catch((err) => {
        console.log(err);
        console.log("deviceID query failed");
});
}
}).then(() => {
const currentTime = Date.now()
```

```
if(redFlag > 4) {
                    //// Reads atleast 5 warnings before
   triggering notification
if(currentTime - lastTriggerTime > (
   minTimeIntervalBetweenNotifications * 1000)) {
        console.log("****warning temp above limit");
        Device.getDeviceID(tempData).then((device) => {
                User.logWarning(device._id, true, "
                   temperature")
        }).catch((err) => {
                console.log(err);
                console.log("deviceID query failed");
        });
        lastTriggerTime = currentTime;
} else {
        console.log("*****not now");
}
}
}).catch((flag) => {
if(flag) {
redFlag = 0;
Device.getDeviceID(tempData).then((device) => {
        User.logWarning(device._id, false, "temperature")
}).catch((err) => {
        console.log(err);
        console.log("deviceID query failed");
});
console.log("****cool");
} else {
console.log("****not so cool");
}
});
Device.logTemperatureData(tempData).then((msg) => {
```

```
console.log(msg, tempData.data);
callback(msg);
}).catch((err) => {
console.log(err);
callback(err);
});
});

socket.on('warnInventry', (invData) => {
console.log(invData.data)
Device.logCurrentWeight(invData).then((device) => {

}).catch(err => {
console.log(err);
})
Device.surveyInventryData(invData).then((flag) => {
if(flag) {
redInvFlag++;
} else {
redInvFlag = 0;
Device.getDeviceID(invData).then((device) => {
User.logWarning(device._id, false, "weight")
}).catch((err) => {
console.log(err);
console.log("deviceID query failed");
});
}
}).then(() => {
const currentTime = Date.now()
if(redInvFlag > 4) {
if(currentTime - lastInvTriggerTime > (
   minTimeIntervalBetweenNotifications * 1000)) {
console.log("****warning inventry below limit");
```

```
Device.getDeviceID(invData).then((device) => {
        User.logWarning(device._id, true, "weight")
}).catch((err) => {
        console.log(err);
        console.log("deviceID query failed");
});
lastInvTriggerTime = currentTime;
} else {
console.log("*****not now");
}

}
}).catch((flag) => {
if(flag) {
redInvFlag = 0;
Device.getDeviceID(invData).then((device) => {
        User.logWarning(device._id, false, "weight")
}).catch((err) => {
        console.log(err);
        console.log("deviceID query failed");
});
console.log("****ok inv");
} else {
console.log("****not ok inv");
}
});
});

socket.to(room).on('imageFetch', (data) => {
socket.to(room).emit('deviceImageFetch');
});

socket.to(room).on('serverMonitorImage', (data) => {
```

```
console.log('******emitting fetched data');
socket.to(room).emit('clientImageFetch', data);
});

socket.to(room).on('temp_control_monitor', (data, deviceID
    ) => {
Device.updateSettings(data, deviceID)
socket.to(room).broadcast.emit('temp_control_monitor',
    data);
});

socket.to(room).on('temp_control_threshold', (data,
    deviceID) => {
Device.updateSettings(data, deviceID)
socket.to(room).broadcast.emit('temp_control_threshold',
    data);
});

socket.to(room).on('inventry_control_monitor', (data,
    deviceID) => {
Device.updateSettings(data, deviceID)
socket.to(room).broadcast.emit('inventry_control_monitor',
     data);
});

socket.to(room).on('inventry_control_threshold', (data,
    deviceID) => {
Device.updateSettings(data, deviceID)
socket.to(room).broadcast.emit('inventry_control_threshold
    ', data);
});

});
```

```
socket.on('disconnect', () => {
console.log('Disconnected from client');
});
});

app.post('/log', (req, res) => {
Device.logTemperatureData(req.body).then(() => {
res.status(200).send();
}).catch(() => {
res.status(400).send();
});
});

/// Add Device to Master Device List, i/p: n/a, payload: {
   serial: '1234567'}     (Done)

app.post('/master', superAuthenticate, (req, res) => {
var body = _.pick(req.body, ['serial']);
var device = new Device({
serial: body.serial
});

device.save().then((details) => {
res.send(details);
}, err => {
res.status(400).send(err);
})
});

/// Add Device to User Device List, i/p: n/a, payload: {
   serial: '1234567'}       (Done)
```

```
app.post('/device', authenticate, (req, res) => {
var body = _.pick(req.body, ['name','serial']);
Device.findOne({
serial: body.serial
}).then((device) => {
return User.verifyDevice(req.user._id, device._id).then(()
    => {
return Promise.resolve();
}).catch(() => {
return req.user.addDevice(device._id, body.name).then((
   devices) => {
return Promise.resolve(devices);
}).catch(() => {
return Promise.reject();
});
});
}).then((devices) => {
if(!devices) {
User.findById(req.user._id).then((user) => {
res.send(user.devices);
}, err => res.status(405).send(err));
} else {
res.send(devices);
}
}).catch((err) => {
res.status(400).send(err);
});
});

/// Get User Device List, i/p: n/a, payload: n/a          (
   Done)

app.get('/device', authenticate, (req, res) => {
```

```
User.findById(req.user._id).then((user) => {
res.send(user.devices);
}, err => res.status(400).send(err));
});

/// Get User Device Details, i/p: deviceID, payload: n/a
            (Done)

app.get('/device/:id', authenticate, (req, res) => {
var id = req.params.id;
User.verifyDevice(req.user._id, id).then(() => {
return Device.findOne({_id: id});
}).then((device) => {
if(!device) return res.status(404).send();
res.send({device});
}).catch(err => res.status(err).send());
});

/// Remove Device from User Device List, i/p: deviceID,
   payload: n/a     (Done)

app.delete('/device/:id', authenticate, (req, res) => {
var id = req.params.id;
if(!ObjectID.isValid(id)) return res.status(404).send();
req.user.removeDevice(id).then(() => {
User.findById(req.user._id).then((user) => {
res.send(user.devices);
}, err => res.status(400).send(err));
}).catch((err) => {
res.status(400).send(err);
});
});
```

```
app.get('/notification/:type/:id', authenticate, (req, res
    ) => {
var entryID = req.params.id;
var type = req.params.type
if(!ObjectID.isValid(entryID)) return res.status(404).send
    ();
User.resetAlert(entryID, type).then(() => {
res.status(200).send('removed alert');
}).catch(err => {
res.status(400).send(err);
});
});

/// Update Device Settings, (Done)

app.patch('/device/pref/:id', authenticate, (req, res) =>
    {
var id = req.params.id;
var body = _.pick(req.body, ['set', 'monitoring', '
    threshold']);

User.verifyDevice(req.user._id, id).then(() => {
return Device.updateSettings(body, id);
}).then((device) => {
res.send(device);
}).catch((err) => {
res.status(err).send();
});
});

/// Create New User Account, i/p: n/a, payload: {email: '
    example@email.com', password: '123456'} (Done)
```

```
app.post('/user/new', (req, res) => {
var body = _.pick(req.body, ['email', 'password']);
var user = new User(body);
user.save().then(() => {
return user.generateAuthToken();
}).then((token) => {
res.header('x-auth', token).send({_id: user._id, email:
    user.email});
}).catch((err) => res.status(400).send(err));
});

/// Get User Object, i/p: n/a, payload: n/a      (Done)


app.get('/user/me', authenticate, (req, res) => {
res.send(req.user);
});

/// Login User, i/p: n/a, payload: {email: 'example@email.
    com', password: '123456'} (Done)

app.post('/user/login', (req, res) => {
var body = _.pick(req.body, ['email', 'password']);
User.findByCredentials(body.email, body.password).then((
    user) => {
return user.generateAuthToken().then((token) => {
res.header('x-auth',token).send({_id: user._id, email:
    user.email});
});
}).catch((err) => {
res.status(400).send(err);
});
});
```

```
/// Logout User, i/p: n/a, payload: n/a (Done)

app.delete('/user/logout', authenticate, (req, res) => {
req.user.removeToken(req.token).then(() => {
res.status(200).send();
}).catch(() => {
res.status(400).send();
});
});

/// Logout All User, i/p: n/a, payload: n/a      (Done)


app.delete('/user/logout/all', authenticate, (req, res) =>
    {
req.user.removeAllToken(req.token).then(() => {
res.status(200).send();
}).catch((err) => {
res.status(400).send(err);
});
});

app.get('/device/self/:serial', (req, res) => {
var data = {
serial : req.params.serial
}
Device.getDeviceID(data).then((device) => {
res.send(device.config);
}).catch((err) => {
res.status(400).send(err);
});
});
```

```
server.listen(port, () => {
console.log(`Listening on port ${port}`);
});

module.exports = {app};
          /// exporting required for testing
```

Listing A.2: authenticate.js

```
var {User} = require('../models/user');
var bcrypt = require('bcrypt');

var authenticate = (req, res, next) => {
var token = req.header('x-auth');
User.findByToken(token).then((user) => {
if(!user) {
        return Promise.reject();
}
req.user = user;
req.token = token;
next();
}).catch((err) => {
res.status(401).send();
});
};

var superAuthenticate = (req, res, next) => {
var keyOne = req.header('x-super-one');
var keyTwo = req.header('x-super-two');
verify(keyOne, keyTwo).then((msg) => {
next();
}).catch((err) => {
res.status(401).send(err);
```

```
});
}

var verify = (one, two) => {
var hashOne = '$2b$10$4uCnOyoxC81YIYLiriPx2O5Tfr.
   CCq0K44eZ8XDWIDmay18Tk5XDy';
var hashTwo = '
   $2b$10$nuxsBYTksm2ZC6r7UV82juefMtuD4M3kTiriEkI0NbEYauXkWJQFm
   ';

return new Promise((resolve, reject) => {
bcrypt.compare(one, hashOne, (err, res) => {
        if(res) {
                bcrypt.compare(two, hashTwo, (err, res) =>
                    {
                        if(res) {
                                resolve('Super
                                   Authentication
                                   Successful');
                        } else reject('Super
                           Authentication Failed');
                });
        } else reject('Super Authentication Failed');
});
});
}

module.exports = {authenticate, superAuthenticate};
```

Listing A.3: pinger.py

```
from threading import Thread
import RPi.GPIO as GPIO
import sys
```

```
from hx711 import HX711
import time
import os
import requests
from socketIO_client_nexus import SocketIO,
   LoggingNamespace

import pigpio
pi = pigpio.pi()

import DHT22
sensor = DHT22.sensor(pi, 17)

hx = HX711(9,11)
hx.set_reading_format("LSB", "MSB")
hx.set_reference_unit(322)
#1 wire 322
#2 wire 311.7

hx.reset()
hx.tare()

enableLog = False
enableInventry = False

def on_connect():
print ('connected to server')

def on_join(*args):
print (args[0])

def on_emit(*args):
print ('got ack from server '+ args[0])
```

```
def on_tempControl(*args):
global enableLog
enableLog = args[0]['monitoring']

def on_inventryControl(*args):
global enableInventry
enableInventry = args[0]['monitoring']

def fetch_Image():
print('sending image from device')
os.system('fswebcam -r 320x240 --no-banner --save /home/pi
   /Final/monitor/image.jpg')
time.sleep(1)
data = open('image.jpg', 'rb').read()
encodedData = data.encode("base64")
socketIO.emit('serverMonitorImage', encodedData)
socketIO.wait(seconds=3)

def cleanAndExit():
print ("Cleaning...")
GPIO.cleanup()
sys.exit()

serial = '1'
baseURL = 'http://192.168.43.11'
port = 4000
url = baseURL + ':' + str(port) + '/device/self/' + serial
config = requests.get(url)
enableLog = config.json()['temperature']['monitoring']
enableInventry = config.json()['weight']['monitoring']

socketIO = SocketIO(baseURL, port, LoggingNamespace)
```

```
socketIO.on('connect',on_connect)
socketIO.emit('join', serial, on_join)
socketIO.wait(seconds=1)
socketIO.on('deviceImageFetch', fetch_Image)
socketIO.wait(seconds=1)

def thread_TempEmit():
while 1 :
if not enableLog:
        print('monitoring off')
        time.sleep(5)
        continue
sensor.trigger()
time.sleep(3)
socketIO.emit('logTemp', {
        'serial': serial,
        'time': time.time(),
        'data': sensor.temperature()/1.0
},on_emit)
socketIO.wait(seconds=2)

def thread_TempMonitor():
while 1:
socketIO.on('temp_control_monitor', on_tempControl)
socketIO.wait(seconds=3)

def thread_InventryMonitor():
while 1:
socketIO.on('inventry_control_monitor', on_inventryControl
   )
socketIO.wait(seconds=3)

def thread_InventryEmit():
```

```
while 1 :
if not enableInventry:
        print('inventry monitoring off')
        time.sleep(5)
        continue
time.sleep(3)
socketIO.emit('warnInventry', {
        'serial': serial,
        'time': time.time(),
        'data': int(round(hx.get_weight(5)))
})
hx.power_down()
hx.power_up()
socketIO.wait(seconds=2)



if __name__ == "__main__":
_thread_temp = Thread(target= thread_TempEmit)
_thread_temp.start()

_thread_tempMon = Thread(target= thread_TempMonitor)
_thread_tempMon.start()

_thread_inventry = Thread(target= thread_InventryEmit)
_thread_inventry.start()

_thread_inventryMon = Thread(target=
   thread_InventryMonitor)
_thread_inventryMon.start()
```

Listing A.4: AppDelegate.swift

```
//
```

```
//  AppDelegate.swift
//  Monitor
//
//  Created by Arjon Das on 9/29/18.
//  Copyright    2018 Arjon Das. All rights reserved.
//

import UIKit
import CoreData

extension UIViewController {
func hideKeyboardWhenTappedAround() {
let tap: UITapGestureRecognizer = UITapGestureRecognizer(
   target: self, action: #selector(UIViewController.
   dismissKeyboard))
tap.cancelsTouchesInView = false
view.addGestureRecognizer(tap)
}

@objc func dismissKeyboard() {
view.endEditing(true)
}
}

@UIApplicationMain

class AppDelegate: UIResponder, UIApplicationDelegate {

var window: UIWindow?
let serial = "1"
let url = Bundle.main.infoDictionary!["Server IP"] as!
   String
let socket = Socket()
```

```
var timer = 0

let mainStoryBoard = UIStoryboard(name: "Main", bundle:
    nil)

var dispatch : DispatchQueue = DispatchQueue.main
var isActive : Bool = true

var viewControllers : [UIViewController]!

var menuViewController : MenuViewController!
var signInViewController : SignInViewController!
var deviceViewController : DeviceViewContoller!
var panelViewController : PanelViewController!
var addDeviceViewController : AddDeviceViewController!
var plotViewController : PlotViewController!

func startAltMenuViewController() {
//          print("********Showing Alternate Menu View
    Controller")
viewControllers.append(menuViewController)
self.window?.rootViewController?.present(
    menuViewController, animated: true, completion: nil)
}

func startAltSignInViewController() {
//          print("Token doesn't exist")
viewControllers.append(signInViewController)
self.window?.rootViewController?.present(
    signInViewController, animated: true, completion: nil)
}

func startAltDeviceViewController() {
```

```
//          print("********Launching Alternate Device List")
viewControllers.append(deviceViewController)
self.window?.rootViewController?.presentedViewController?.
    present(deviceViewController, animated: true,
    completion: nil)
}

func startAltPanelViewController(name: String, id: String)
     {
//          print("*********Showing Alternate Details View
    Controller")
panelViewController.deviceID = id
panelViewController.deviceName = name
viewControllers.append(panelViewController)
deviceViewController.present(panelViewController, animated
    : true, completion: nil)
}

func startAltAddDeviceViewController() {
//          print("Launching Add Device")
viewControllers.append(addDeviceViewController)
self.window?.rootViewController?.presentedViewController?.
    present(addDeviceViewController, animated: true,
    completion: nil)
}

func startAltPlotViewController(name: String, id: String)
    {
plotViewController.deviceID = id
plotViewController.deviceName = name
viewControllers.append(plotViewController)
panelViewController.present(plotViewController, animated:
    true, completion: nil)
```

```
}

func dismiss(viewController: UIViewController) {
viewController.dismiss(animated: true, completion: nil)
viewControllers.removeLast()
}

func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [
    UIApplicationLaunchOptionsKey: Any]?) -> Bool {
viewControllers = [UIViewController]()
menuViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    MenuViewController") as! MenuViewController
signInViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    SignInViewController") as! SignInViewController
deviceViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    DeviceViewController") as! DeviceViewContoller
panelViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    PanelViewController") as! PanelViewController
addDeviceViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    AddDeviceViewController") as! AddDeviceViewController
plotViewController = mainStoryBoard.
    instantiateViewController(withIdentifier: "
    PlotViewController") as! PlotViewController

let notificationSettings = UIUserNotificationSettings(
    types: [.alert, .sound], categories: nil)
UIApplication.shared.registerUserNotificationSettings(
```

```
    notificationSettings)
return true
}

func application(_ application: UIApplication, didReceive
    notification: UILocalNotification) {
self.takeActionWithNotification(notification: notification
    )
}

func takeActionWithNotification(notification:
    UILocalNotification) {
//        let lastViewController = viewControllers.last
//        var message = ""
//        if (lastViewController?.isKind(of:
    UIAlertController.self))! {
//            message = (lastViewController as!
    UIAlertController).message! + "\n" + notification.
    alertBody!
//            dismiss(viewController: lastViewController!)
//        } else {
//            message = notification.alertBody!
//        }
//        let alertController = UIAlertController(title: "
    Alert!!", message: message, preferredStyle: .alert)
//        let dismissAction = UIAlertAction(title: "
    Dismiss", style: .default, handler: nil)
//        alertController.addAction(dismissAction)
//        viewControllers.last?.present(alertController,
    animated: true, completion: nil)
////          self.window?.rootViewController?.
    presentedViewController?.present(alertController,
    animated: true, completion: nil)
```

```
let alertViewController = UIAlertController(title: "Alert
    !!!", message: notification.alertBody, preferredStyle:
    .alert)
let dismissAction = UIAlertAction(title: "Dismiss", style:
     .default, handler: nil)
alertViewController.addAction(dismissAction)

let alertWindow = UIWindow(frame: UIScreen.main.bounds)
alertWindow.rootViewController = UIViewController()
alertWindow.windowLevel = UIWindowLevelAlert + 1
alertWindow.makeKeyAndVisible()
alertWindow.rootViewController?.present(
    alertViewController, animated: true, completion: nil)
}

func triggerNotification(msg : (String, String)) {
let localNotification = UILocalNotification()
localNotification.fireDate = NSDate(timeIntervalSinceNow:
    5) as Date
localNotification.applicationIconBadgeNumber = 1
localNotification.soundName =
    UILocalNotificationDefaultSoundName
localNotification.userInfo = [
"message": "Device Alert!!"
]
localNotification.alertBody = "\(msg.0)! Devices: \(msg.1)
    "
UIApplication.shared.scheduleLocalNotification(
    localNotification)
}

func applicationWillResignActive(_ application:
    UIApplication) {
```

```
// Sent when the application is about to move from active
   to inactive state. This can occur for certain types of
   temporary interruptions (such as an incoming phone call
    or SMS message) or when the user quits the application
    and it begins the transition to the background state.
// Use this method to pause ongoing tasks, disable timers,
    and invalidate graphics rendering callbacks. Games
   should use this method to pause the game.
}

func applicationDidEnterBackground(_ application:
   UIApplication) {
// Use this method to release shared resources, save user
   data, invalidate timers, and store enough application
   state information to restore your application to its
   current state in case it is terminated later.
// If your application supports background execution, this
    method is called instead of applicationWillTerminate:
   when the user quits.
isActive = false
application.beginBackgroundTask(withName: "pinging",
   expirationHandler: nil)
performSelector(inBackground: #selector(pinging), with:
   nil)
}

@objc func pinging() {
let defaults = UserDefaults.standard
let token = defaults.object(forKey: "x-auth") as? String
if token != nil {
print("background pinging")
socket.pinging()
}
```

```
if !isActive {
sleep(6)
performSelector(inBackground: #selector(pinging), with:
    nil)
} else {
return
}
}

@objc func checking() {
let defaults = UserDefaults.standard
let token = defaults.object(forKey: "x-auth") as? String
if token != nil {
print("foreground pinging")
socket.pinging()
}
if isActive {
dispatch.asyncAfter(deadline: .now() + .seconds(6),
    execute: {
     self.perform(#selector(self.checking))
})
} else {
return
}
}

func applicationWillEnterForeground(_ application:
    UIApplication) {
// Called as part of the transition from the background to
     the active state; here you can undo many of the
    changes made on entering the background.
isActive = true
}
```

```
func applicationDidBecomeActive(_ application:
    UIApplication) {
// Restart any tasks that were paused (or not yet started)
     while the application was inactive. If the application
     was previously in the background, optionally refresh
    the user interface.
perform(#selector(checking))
}

func applicationWillTerminate(_ application: UIApplication
    ) {
// Called when the application is about to terminate. Save
     data if appropriate. See also
    applicationDidEnterBackground:.
// Saves changes in the application's managed object
    context before the application terminates.
self.saveContext()
}

// MARK: - Core Data stack

@available(iOS 10.0, *)
lazy var persistentContainer: NSPersistentContainer = {
/*
The persistent container for the application. This
    implementation
creates and returns a container, having loaded the store
    for the
application to it. This property is optional since there
    are legitimate
error conditions that could cause the creation of the
    store to fail.
```

```
*/
let container = NSPersistentContainer(name: "Monitor")
container.loadPersistentStores(completionHandler: { (
   storeDescription, error) in
if let error = error as NSError? {
    // Replace this implementation with code to handle the
        error appropriately.
    // fatalError() causes the application to generate a
       crash log and terminate. You should not use this
       function in a shipping application, although it may
        be useful during development.

    /*
        Typical reasons for an error here include:
        * The parent directory does not exist, cannot be
           created, or disallows writing.
        * The persistent store is not accessible, due to
           permissions or data protection when the device
           is locked.
        * The device is out of space.
        * The store could not be migrated to the current
           model version.
        Check the error message to determine what the
           actual problem was.
        */
    fatalError("Unresolved error \(error), \(error.
       userInfo)")
}
})
return container
}()

// MARK: - Core Data Saving support
```

```
func saveContext () {
if #available(iOS 10.0, *) {
let context = persistentContainer.viewContext
if context.hasChanges {
    do {
        try context.save()
    } catch {
        // Replace this implementation with code to handle
            the error appropriately.
        // fatalError() causes the application to generate
            a crash log and terminate. You should not use
           this function in a shipping application,
           although it may be useful during development.
        let nserror = error as NSError
        fatalError("Unresolved error \(nserror), \(nserror
           .userInfo)")
    }
}
} else {
// Fallback on earlier versions
}

}

}
```

Listing A.5: Socket.swift

```
//
//  Socket.swift
//  Monitor
//
//  Created by Arjon Das on 10/12/18.
```

```
//  Copyright    2018 Arjon Das. All rights reserved.
//
import UIKit
import SocketIO
import Alamofire

class Socket {
let manager = SocketManager(socketURL: URL(string: Bundle.
   main.object(forInfoDictionaryKey: "Server IP") as!
   String)!, config: [.log(false), .compress])
var socket : SocketIOClient
init () {
socket = manager.defaultSocket
}

func Connect() {
socket.on(clientEvent: .connect) { data, ack in
self.socket.emitWithAck("join", Bundle.main.object(
   forInfoDictionaryKey: "Serial") as! String).timingOut(
   after: 1) { data in
print(data[0])
}
}
socket.connect()
}

func ListenToControlTempSwitch(toggle : UISwitch) {
socket.on("temp_control_monitor") { (data, ack) in
let controls = data[0] as! Dictionary <String, Any>
if controls["set"] as! String == "temperature" {
toggle.setOn(controls["monitoring"] as! Bool, animated:
   true)
}
```

```
}
}

func ListenToControlTempLimit(textField : UITextField) {
socket.on("temp_control_threshold") { (data, ack) in
let controls = data[0] as! Dictionary <String, Any>
if controls["set"] as! String == "temperature" {
textField.text = String(controls["threshold"] as! Float)
}
}
}

func ListenToControlInventrySwitch(toggle : UISwitch) {
socket.on("inventry_control_monitor") { (data, ack) in
let controls = data[0] as! Dictionary <String, Any>
if controls["set"] as! String == "weight" {
toggle.setOn(controls["monitoring"] as! Bool, animated:
   true)
}
}
}

func ListenToControlInventryLimit(textField : UITextField)
    {
socket.on("inventry_control_threshold") { (data, ack) in
let controls = data[0] as! Dictionary <String, Any>
if controls["set"] as! String == "weight" {
textField.text = String(controls["threshold"] as! Float)
}
}
}

func EmitTempToggle(deviceID: String, value: Bool) {
```

```
let payload : Dictionary <String, Any> = [
"set": "temperature",
"monitoring": value
]
socket.emit("temp_control_monitor", payload, deviceID)
}

func EmitTempLimit(deviceID: String, value: Float) {
let payload : Dictionary <String, Any> = [
"set": "temperature",
"threshold": value
]
socket.emit("temp_control_threshold", payload, deviceID)
}

func EmitInventryToggle(deviceID: String, value: Bool) {
let payload : Dictionary <String, Any> = [
"set": "weight",
"monitoring": value
]
socket.emit("inventry_control_monitor", payload, deviceID)
}

func EmitInventryLimit(deviceID: String, value: Float) {
let payload : Dictionary <String, Any> = [
"set": "weight",
"threshold": value
]
socket.emit("inventry_control_threshold", payload,
   deviceID)
}

func pinging() {
```

```
getDeviceWarningData(completion: {(temp, weight) in
if temp != "" {
print("triggering temp notification")
self.triggerNotification(msg: ("Temperature Alert", temp))
}
if weight != "" {
print("triggering weight notification")
self.triggerNotification(msg: ("Inventory Alert", weight))
}
})
}

func triggerNotification(msg : (String, String)) {
let localNotification = UILocalNotification()
localNotification.fireDate = NSDate(timeIntervalSinceNow:
    2) as Date
localNotification.soundName =
    UILocalNotificationDefaultSoundName
localNotification.userInfo = [
"message": "Test notification msg"
]
localNotification.alertBody = "\(msg.0)! Devices: \(msg.1)
    "
UIApplication.shared.scheduleLocalNotification(
    localNotification)
}

func resetNotification(for entryID : String, type : String
    , headers: HTTPHeaders) {
let url : String = Bundle.main.object(forInfoDictionaryKey
    : "Server IP") as! String + "notification/" + type +
    "/" + entryID + "/"
print(url)
```

```
Alamofire.request(url, method: .get, headers: headers).
    validate().responseString { response in
switch response.result {
case .success:
print("Alert Reset Successful")
case .failure(let err):
print("Alert Reset Not Successful, might alert again")
print(err)
}
}
}

func getDeviceWarningData(completion:@escaping ((String,
    String)) -> Void) {
let defaults = UserDefaults.standard
let token = defaults.object(forKey: "x-auth") as? String
if token == nil {
return
}
let headers : HTTPHeaders = [
"x-auth": token!
]
let url : String = Bundle.main.object(forInfoDictionaryKey
    : "Server IP") as! String + "device/"
var targetDevicesTemp : String = ""
var targetDevicesWeight : String = ""
Alamofire.request(url, method: .get, headers: headers).
    validate().responseJSON { response in
switch response.result {
case .success:
if let responseObject = response.result.value {
    let deviceList : [Dictionary] = responseObject as! [
        Dictionary<String, Any>]
```

```
        for device in deviceList {
            let name = device["_name"] as! String
            let entryID = device["_id"] as! String
            let warnTemp = device["warnTemp"] as! Bool
            let warnWeight = device["warnWeight"] as! Bool
            if warnTemp {
                if targetDevicesTemp == "" {
                    targetDevicesTemp = targetDevicesTemp +
                        name
                } else {
                    targetDevicesTemp = targetDevicesTemp + ",
                        \(name)"
                }
                self.resetNotification(for: entryID, type: "
                    temperature", headers: headers)
            }
            if warnWeight {
                if targetDevicesWeight == "" {
                    targetDevicesWeight = targetDevicesWeight
                        + name
                } else {
                    targetDevicesWeight = targetDevicesWeight
                        + ", \(name)"
                }
                self.resetNotification(for: entryID, type: "
                    weight", headers: headers)
            }
        }
    }
    completion((targetDevicesTemp, targetDevicesWeight))
    break
    case .failure:
    targetDevicesTemp = ""
```

```
targetDevicesWeight = ""
completion((targetDevicesTemp, targetDevicesWeight))
break
}
}
}

func fetchImage(imageView: UIImageView, loading:
    UIActivityIndicatorView) {
loading.startAnimating()
socket.emit("imageFetch", true)
socket.on("clientImageFetch") { (data, ack) in
self.socket.off("clientImageFetch")
let rawData = data[0] as! String
let imageData = NSData(base64Encoded: rawData, options: .
    ignoreUnknownCharacters)
let image = UIImage(data: imageData! as Data)
imageView.contentMode = .scaleAspectFit
imageView.image = image
imageView.backgroundColor = UIColor.init(red: 0, green: 0,
     blue: 0, alpha: 0.5)
imageView.isHidden = false
loading.stopAnimating()
}
let dispatch : DispatchQueue = DispatchQueue.main
dispatch.asyncAfter(deadline: .now() + .seconds(10),
    execute: {
loading.stopAnimating()
self.socket.off("clientImageFetch")
})
}
}
```

Listing A.6: PanelViewController.swift

```
//
//  PanelViewController.swift
//  Monitor
//
//  Created by Arjon Das on 10/3/18.
//  Copyright    2018 Arjon Das. All rights reserved.
//

import UIKit
import Alamofire
import SocketIO

class PanelViewController: UIViewController,
   UITextFieldDelegate {
let appDelegate = UIApplication.shared.delegate as!
   AppDelegate
let socket = Socket()
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var loading: UIActivityIndicatorView!



@IBAction func tempLimit(_ sender: Any) {
let obj = sender as AnyObject
let inputField = obj as! UITextField
let input : String? = inputField.text
if let inputValue = input, !(input?.isEmpty)! {
let value = Float(inputValue)
socket.EmitTempLimit(deviceID: deviceID, value: (value)!)
}
}

@IBAction func plotTempData(_ sender: Any) {
(UIApplication.shared.delegate as! AppDelegate).
```

```
    startAltPlotViewController(name: deviceName, id:
    deviceID)
}

@IBOutlet weak var foodImage: UIImageView!

@IBOutlet weak var _tempLimit: UITextField!

@IBAction func invLimit(_ sender: Any) {
let obj = sender as AnyObject
let inputField = obj as! UITextField
let input : String? = inputField.text
if let inputValue = input, !(input?.isEmpty)! {
let value = Float(inputValue)
socket.EmitInventryLimit(deviceID: deviceID, value: (value
    )!)
}
}

@IBOutlet weak var _invLimit: UITextField!

@IBAction func tempSwitch(_ sender: Any) {
let button = sender as AnyObject
if button.isOn {
socket.EmitTempToggle(deviceID: deviceID, value: true)
} else {
socket.EmitTempToggle(deviceID: deviceID, value: false)
}
}

@IBOutlet weak var _tempSwitch: UISwitch!

@IBAction func invSwitch(_ sender: Any) {
```

```
let button = sender as AnyObject
if button.isOn {
socket.EmitInventryToggle(deviceID: deviceID, value: true)
} else {
socket.EmitInventryToggle(deviceID: deviceID, value: false
    )
}
}

@IBOutlet weak var _invSwitch: UISwitch!

@IBAction func cameraLoad(_ sender: Any) {
socket.fetchImage(imageView: foodImage, loading: loading)
}

@IBAction func back(_ sender: Any) {
//         self.dismiss(animated: true, completion: nil)
print("hello i was dismissed")
(UIApplication.shared.delegate as! AppDelegate).dismiss(
    viewController: self)
}

var deviceID = ""
var deviceName = ""
var tokenGlobal = ""

override func viewWillAppear(_ animated: Bool) {
super.viewWillAppear(animated)
nameLabel.numberOfLines = 2
foodImage.isHidden = true
foodImage.isUserInteractionEnabled = true
loading.hidesWhenStopped = true
loading.stopAnimating()
```

```
self.hideKeyboardWhenTappedAround()
_tempLimit.delegate = self
_invLimit.delegate = self
nameLabel.text = deviceName
hideImageWhenTappedAround()
let defaults = UserDefaults.standard
let token = defaults.object(forKey: "x-auth") as? String
if token == nil {
(UIApplication.shared.delegate as! AppDelegate).dismiss(
    viewController: self)
} else {
tokenGlobal = (token)!
}
loadControls()
socket.Connect()
socket.ListenToControlTempSwitch(toggle: _tempSwitch)
socket.ListenToControlTempLimit(textField: _tempLimit)
socket.ListenToControlInventrySwitch(toggle: _invSwitch)
socket.ListenToControlInventryLimit(textField: _invLimit)
//          socket.pinging()

}

//     override func viewDidAppear(_ animated: Bool) {
//         super.viewDidAppear(animated)
//
//     }

func hideImageWhenTappedAround() {
let tap : UIGestureRecognizer = UITapGestureRecognizer(
    target: self, action: #selector(hideImage))
tap.cancelsTouchesInView = false
foodImage.addGestureRecognizer(tap)
```

```
}

@objc func hideImage() {
foodImage.isHidden = true
}

func textFieldShouldReturn(_ textField: UITextField) ->
    Bool {
textField.resignFirstResponder()
return true
}

func loadControls() {
let url : String = (UIApplication.shared.delegate as!
    AppDelegate).url + "device/" + deviceID
let headers : HTTPHeaders = [
"x-auth": tokenGlobal
]
Alamofire.request(url, method: .get, headers: headers).
    validate().responseJSON { response in
switch response.result {
case .success:
if let responseObject = response.result.value {
let devices : Dictionary = responseObject as! Dictionary<
    String,Any>
let device : Dictionary = devices["device"] as! Dictionary
    <String,Any>
let currentTemp : CGFloat = device["currentTemp"] as!
    CGFloat
let currnetWeight : CGFloat = device["currentWeight"] as!
    CGFloat
let config : Dictionary = device["config"] as! Dictionary<
    String,Any>
```

```
let weight : Dictionary = config["weight"] as! Dictionary<
    String,Any>
let temperature : Dictionary = config["temperature"] as!
    Dictionary<String,Any>
let _tempMonitor : Bool = temperature["monitoring"] as!
    Bool
let _tempVal : Float = temperature["threshold"] as! Float
let _weightMonitor : Bool = weight["monitoring"] as! Bool
let _weightVal : Float = weight["threshold"] as! Float
self.nameLabel.text = "Temp: \(currentTemp.precision(1)!)
     C \n Weight: \(currnetWeight.precision(1)!) gm"
self._tempLimit.text = String(_tempVal)
self._invLimit.text = String(_weightVal)
self._tempSwitch.isOn = _tempMonitor
self._invSwitch.isOn = _weightMonitor
}
case .failure:
print("Error")
}
}
}
}
```