

Missing Scoop

A Project Report
Presented to
The Faculty of the Computer Engineering Department
San Jose State University



San José State
UNIVERSITY

In Partial Fulfillment
Of the Requirements for the Degree Master of Science in Software Engineering

Submitted To
ChandraSekar Vuppalapati

Submitted By
Anushri Srinath Aithal, Ashwini Shankar Narayan, Mojdeh Keykhanzadeh

May 16, 2018

Table of Contents

Chapter 1. Introduction.....	2
1.1. Project Description and Overview.....	2
1.2. Requirements.....	3
1.3. References.....	4
Chapter 2. System Design.....	5
2.1. System Architecture High Level Design	5
2.2. System Component Level Design.....	6
2.3. System Design Pattern Description.....	7
2.4 Mobile UI Design and Wireframes	11
2.5. System Workflow	16
Chapter 3. System Implementation.....	17
3.1. Hardware System Implementation Summary	17
3.2. Mobile and Cloud Technologies Description.....	23
3.3 System Interface and Server Side Design.....	38
3.4 Client Side Design.....	38
Chapter 4. System Testing and Experiment.....	58
4.1. Testing Scope.....	58
4.2. Automation Testing.....	58
4.3. Manual Testing Scenarios.....	64
4.4 Profiling.....	65
Chapter 5. Conclusion and Future Work.....	65
5.1. Project Summary.....	65
5.2. Individual Contributions.....	65
5.3. Project Deployment Instructions.....	66
5.4. Future Work	67

1. Introduction

1.1 Overview

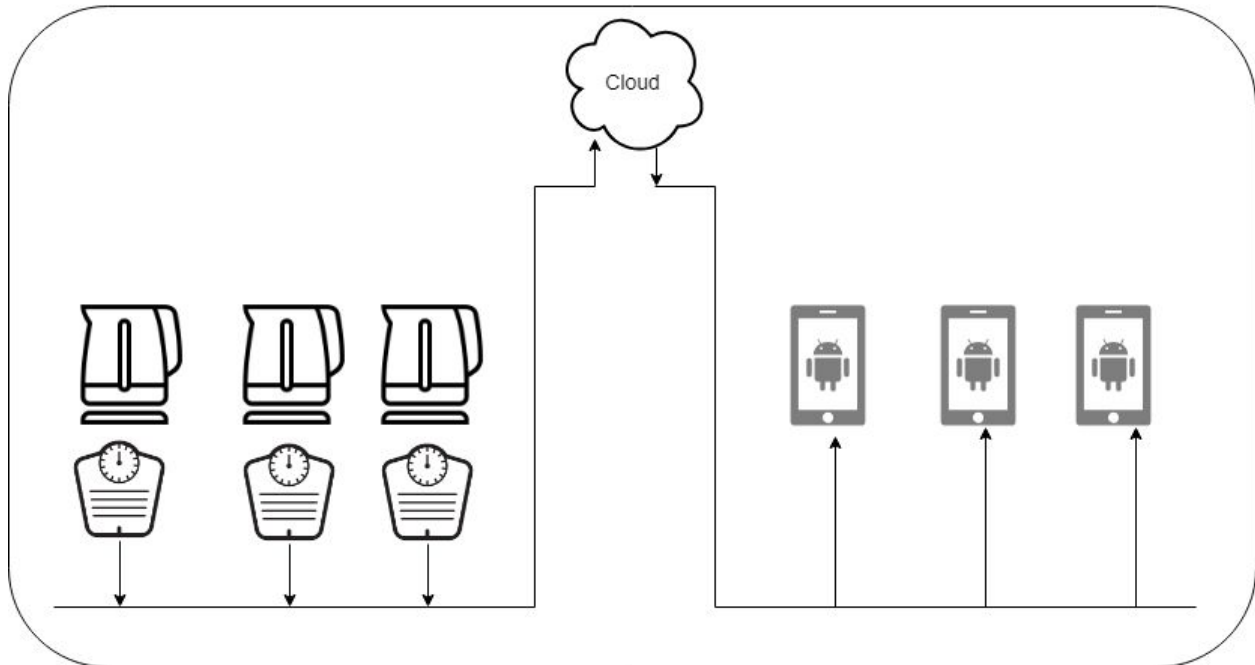
Advance in Internet of Things and Mobile Applications has opened doors for Smart Appliances like Smart Agriculture, Smart Security Systems. Smart Kitchen appliances is the new focus in Internet of Things. With sensors, microcontrollers, mobile applications kitchen appliances can be made more intelligent and responsive.

Missing Scoop incorporates sensors, mobile application, cloud technologies, grocery stores and grocery items to provide a smart inventory tracking, nutrition tracking and facilitate an intelligent system.

Busy lifestyles have made it hard to manage grocery stocking and shopping. Elderly parents also find it hard to make multiple errands to shop groceries. On the other hand, tracking food consumption for babies to maintain a healthy diet and healthy development would involve recording continuous food consumption. With no proper insights into grocery at home elderly people have to go around multiple times to stock up groceries. Lack of proper planning and no information on consumption pattern results in visiting grocery stores multiple times for buying items. Eventually, it becomes a tedious task to observe family consumption, refill interval and stock required to cater guest visits.

The solution to above mentioned problem is having smart containers that track grocery consumption and continuously provide insights into kitchen cabinets. We propose a system where containers are equipped with smart devices using load sensors and ESP32 microcontroller which captures the weight of the container continuously and uploads that data to cloud with the help of MQTT client and AWS IoT configuration. This weight states of the container is monitored to identify if the grocery weight goes below a threshold and notify the user about the time to purchase. The application also provides visualization of current weight of the grocery and enables easy tracking. Furthermore, the application also houses a nutrition tracking system that encourages healthy eating habits. Based on historical data and daily consumption a nutrition pattern can be observed and users can make eat food in moderation. This nutrition tracking also helps monitoring malnutrition in children and tracking unhealthy eating habits in adults.

The application also provides users with a grocery shopping list where they can add items that is not monitored via the smart containers. Users are also provided with a way to identify grocery stores nearby using google maps.



Overview of Missing Scoop

1.2 Requirements

Functional Requirements

1. User Registration with email and password
2. User Login with email and password
3. Allow users to map their IoT devices to specific product
4. Allow users to track their grocery
5. Continuous monitoring of nutrition consumption
6. Historical nutrition consumption tracking
7. Allow users to add or remove grocery from to shopping list
8. Allow users to view nearby grocery stores
9. Users are notified about grocery weight reaching below a configured threshold
10. Allow users to logout

Hardware Requirements

1. Load cell (weight sensor)
2. HX711
3. ESP32 Microcontroller
4. WiFi service
5. Power supply

Software Requirements

1. Android Studio 3.0.1
2. Android KitKat
3. Android API Level 27
4. Eclipse
5. Java 8
6. Spring Boot Version 2.0.0
7. AWS Cloud Platform - EC2, Dynamodb, AWS IoT

Testing Requirements

1. Eclipse
2. Java 8
3. TestNg
4. Maven

1.3 References

1. <https://www.androidtutorialpoint.com/networking/android-volley-tutorial/>
2. <https://medium.com/@developine/how-to-send-firebase-push-notifications-from-server-tutorial-3f3e3a78f9a5>
3. <https://code.tutsplus.com/tutorials/reading-qr-codes-using-the-mobile-vision-api--cms-24680>
4. <https://www.androidhive.info/2016/06/android-getting-started-firebase-simple-login-registration-auth/>
5. <https://www.youtube.com/watch?v=OzY2SwKOxbk>
6. <https://docs.aws.amazon.com/iot/latest/developerguide/iot-ddb-rule.html>
7. https://exploreembedded.com/wiki/Secure_IOT_with_AWS_and_Hornbill_ESP32
8. https://exploreembedded.com/wiki/AWS_IOT_with_Arduino_ESP32

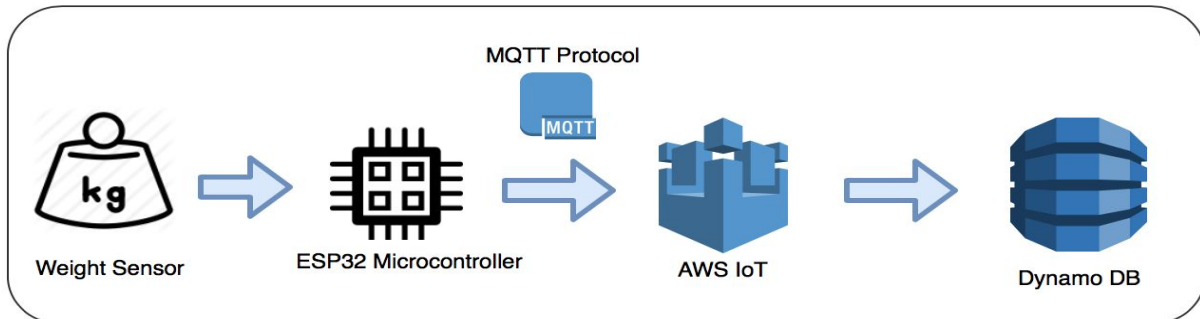
2. System Design

2.1 System High Level Architecture Design

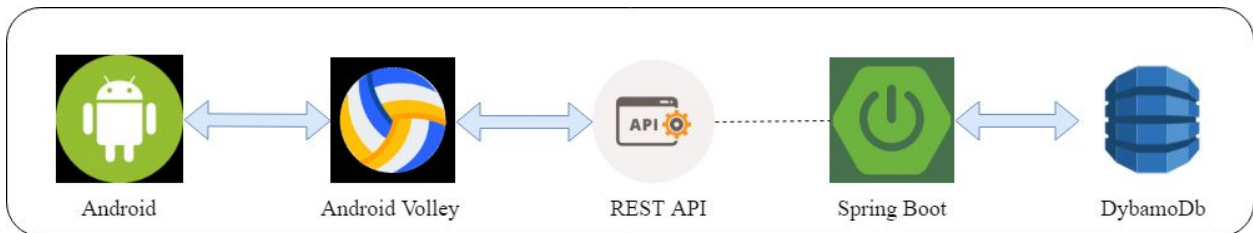


2.2 System Component Level Architecture Design

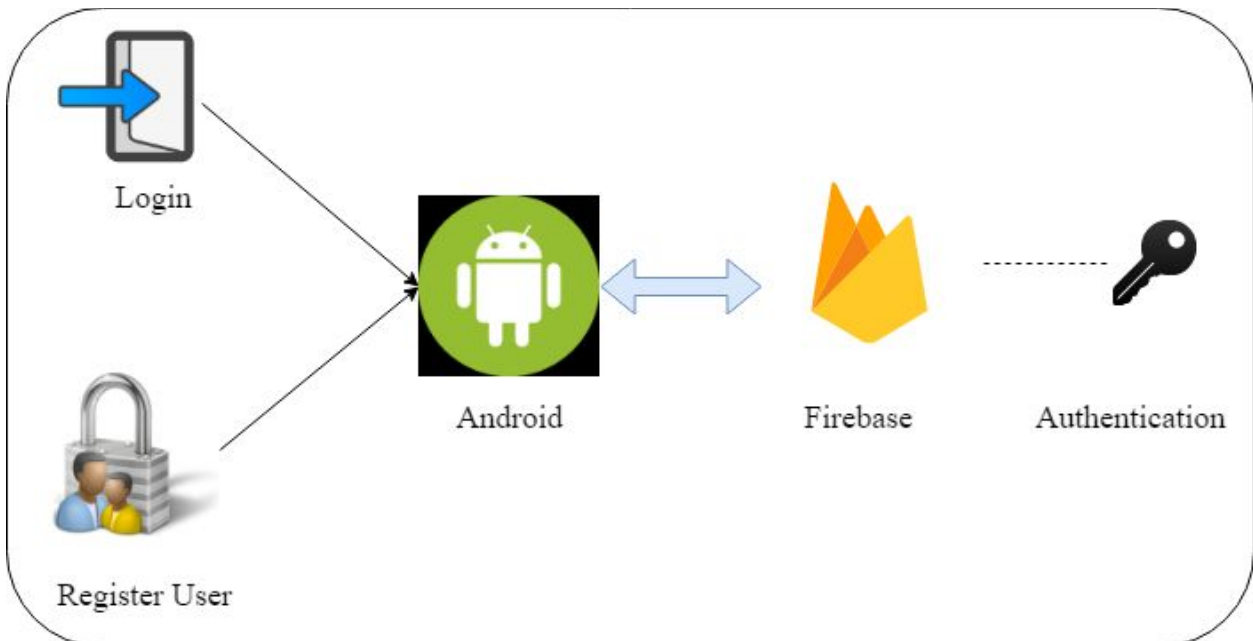
2.2.1 IoT system Design



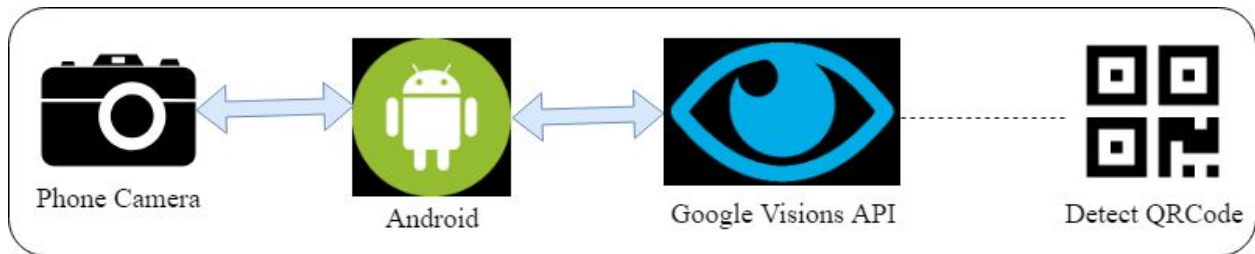
2.2.2 Rest API Level Design



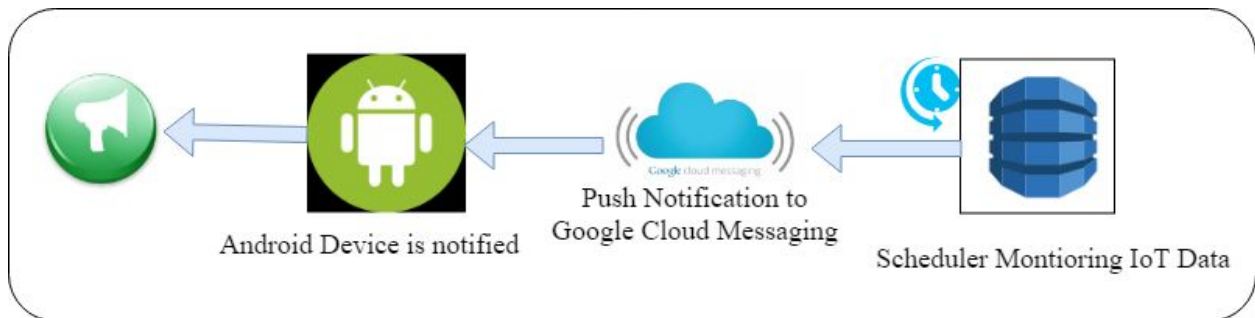
2.2.3 Authentication Component Design



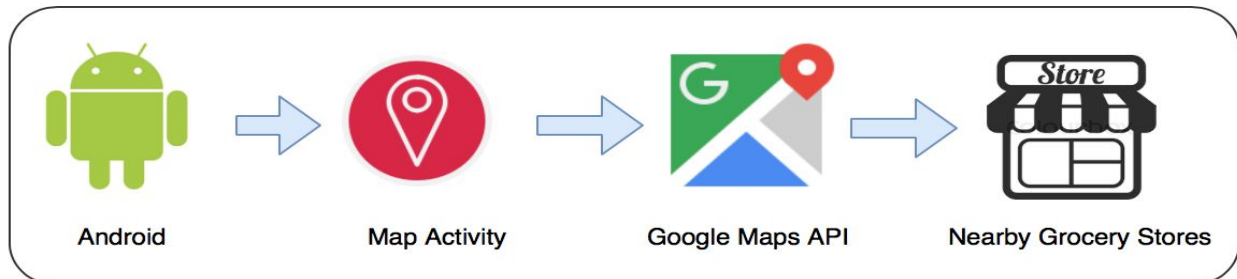
2.2.4 QRCode Scan Component Design



2.2.4 Push Notification Component Design



2.2.5 Google Maps Component Design



2.3 System Design Pattern Description

Backend, Cloud and Database

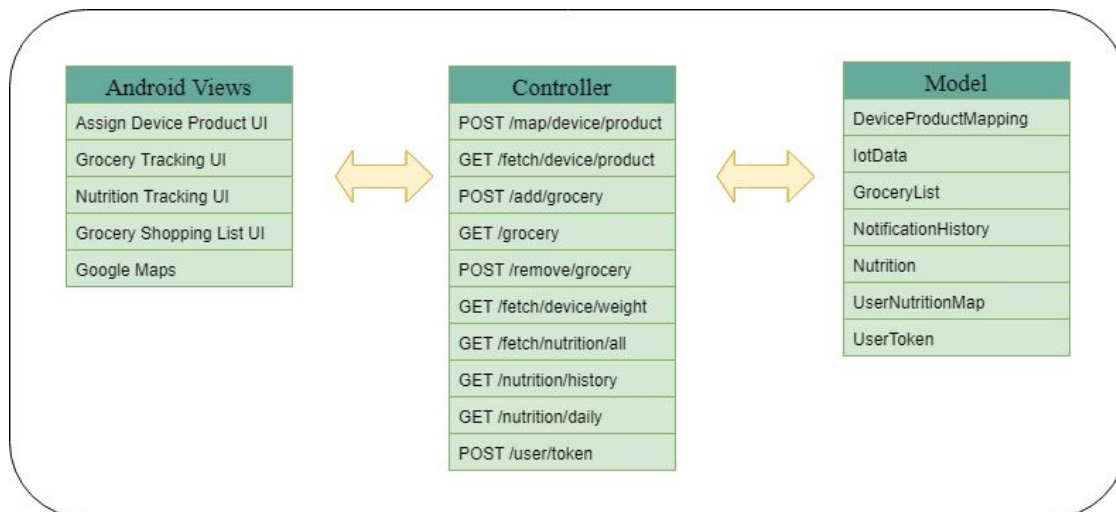
The project is developed adhering to all the Enterprise Integration Design Patterns. Primarily, MVC design pattern is used across the project.

The application exposes a series of RESTful APIs that are written in Java using Spring Boot. The APIs form a strong backend system. These APIs can be consumed by Web, iOS, Android or any interface. In our project the APIs are consumed by Android application which provides an intuitive UI for the end users. Further, the APIs are also developed using MVC pattern.

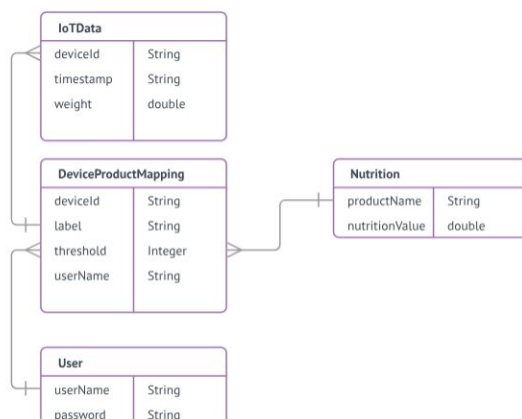
Appropriate usage of abstraction and Object Oriented Programming (OOPs) design principles are upheld. The entire backend application can be divided into 4 major layers

1. Controller Layer: Exposes all the APIs which can be invoked via HTTP methods. Have both POST and GET APIs.
2. Service Layer: Actual business logic is written in this layer. It follows OOPs principles of abstraction. These are spawned by Spring dependency injection.
3. Data Access Layer(DAO): This layer is an abstraction for executing Dynamodb queries. Object Relational Mapping(ORM) principles are used where in the query methods are exposed via an interface. The interface is implemented to fire the actual dynamodb queries. AWS SDK is used to execute the actual queries.
4. Model: Each model represent a physical database table. All operations are done using the model. This follows ORM principles.

The class diagram for the entire backend design can be visualized as below



The Dynamodb tables can be represented as below



The backend is deployed on AWS EC2 servers. The jar runs on an embedded tomcat which comes as a part of Spring Boot. To execute the jar on EC2 we have to install a Java Runtime environment. Java 8 is used for this purpose. Once Java is installed on EC2 the jar can be executed as below

```
java -jar missing-scoop.jar
```

The RESTful APIs developed are consumed by Android application using Volley library. Dependency for volley is added in gradle build as below

```
compile 'com.android.volley:volley:1.0.0'
```

A RestClient Interface is developed which implements GET and POST API calls using Volley. This common framework is used across all API calls in the entire project. The framework developed is as below

```
package edu.sjsu.missing.scoop.api.client;

import android.content.Context;
import android.util.Log;

import com.android.volley.Request;
import com.android.volley.Response;
import com.android.volley.VolleyError;
import com.android.volley.VolleyLog;
import com.android.volley.toolbox.JsonObjectRequest;
import com.google.gson.Gson;

import org.json.JSONException;
import org.json.JSONObject;

import java.util.Map;

import edu.sjsu.missing.scoop.api.request.DeviceProductMappingRequest;

/**
 * Created by Shriaithal on 4/24/2018.
 */

public class RestApiClient {

    private final String BASE_URL = "http://192.168.1.109:8080";
    // "http://10.0.2.2:8080";
    // "http://192.168.0.33:8080"; // "http://18.221.192.106:8080";

    public void executePostAPI(Context context, String uri, JSONObject jsonObject,
    final VolleyAPICallback callback) {
        APIRequestQueue queue = APIRequestQueue.getInstance(context);
        JsonObjectRequest jsonObjectRequest = new
        JsonObjectRequest(Request.Method.POST, BASE_URL + uri, jsonObject, new
        Response.Listener<JSONObject> () {
```

```

        @Override
        public void onResponse(JSONObject jsonResponse) {
            callback.onSuccess(jsonResponse);
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) {
            VolleyLog.e("RestApiClient", "Error: " + error.getMessage());
            callback.onError(error.getMessage());
        }
    });
    queue.getRequestQueue().add(jsonObjectRequest);
}

public void executeGetAPI(Context context, String uri, final VolleyAPICallback
callback) {
    APIRequestQueue queue = APIRequestQueue.getInstance(context);
    JSONObjectRequest jsonObjectRequest = new
    JSONObjectRequest(Request.Method.GET, BASE_URL + uri, null, new
    Response.Listener<JSONObject> () {

        @Override
        public void onResponse(JSONObject jsonResponse) {
            callback.onSuccess(jsonResponse);
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) {
            VolleyLog.e("RestApiClient", "Error: " + error.getMessage());
            callback.onError(error.getMessage());
        }
    }
    ));
    queue.getRequestQueue().add(jsonObjectRequest);
}
}

```

Frontend and Mobile

Android User Interface is the face of the project. User can perform CRUD operations through Android app. Android UI components are used to render a seamless user experience. Key Android features used include

1. Android Activity: Handle invoking REST APIs for user interaction with the application
2. Android Service: Handle asynchronous push notifications and displaying them to user by processing in the background.
3. Android Intent: Used for navigating between screens and carrying information from one screen to other
4. Android AsyncTask: Used for handling Google map features in the application
5. Android Layouts: Constraint Layout, Relative Layout, CardView, RecyclerView, ListViews
6. Android Components: Buttons, Imagebutton, Floating Button, Action Bars

7. Google Map, Android Phone Camera, Android Notification

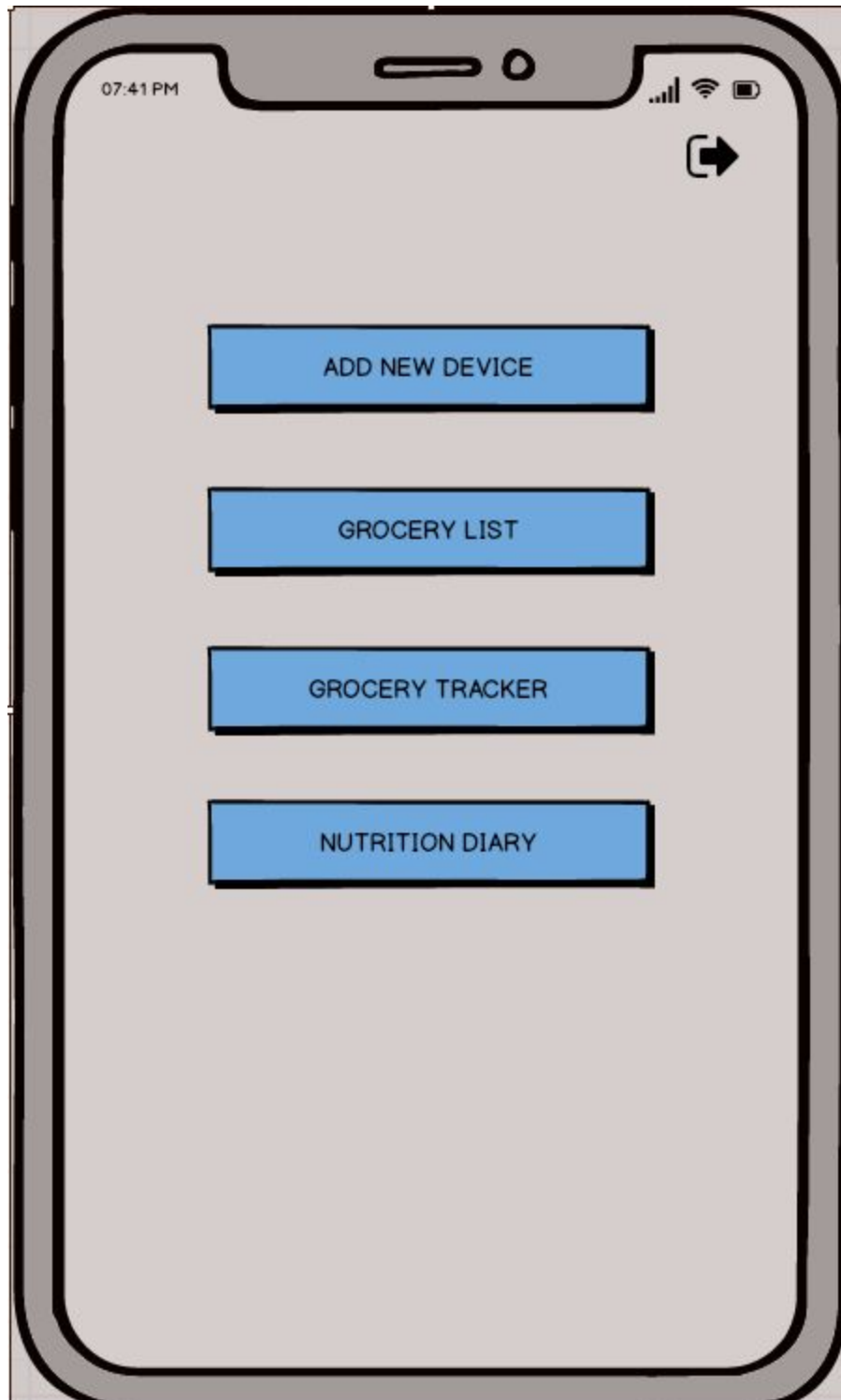
2.4 Mobile UI Wireframes

Balsamiq is used to design our app screen . Below are wireframes for our screens.

Login:



Menu :



Add New Device :

07:43 PM

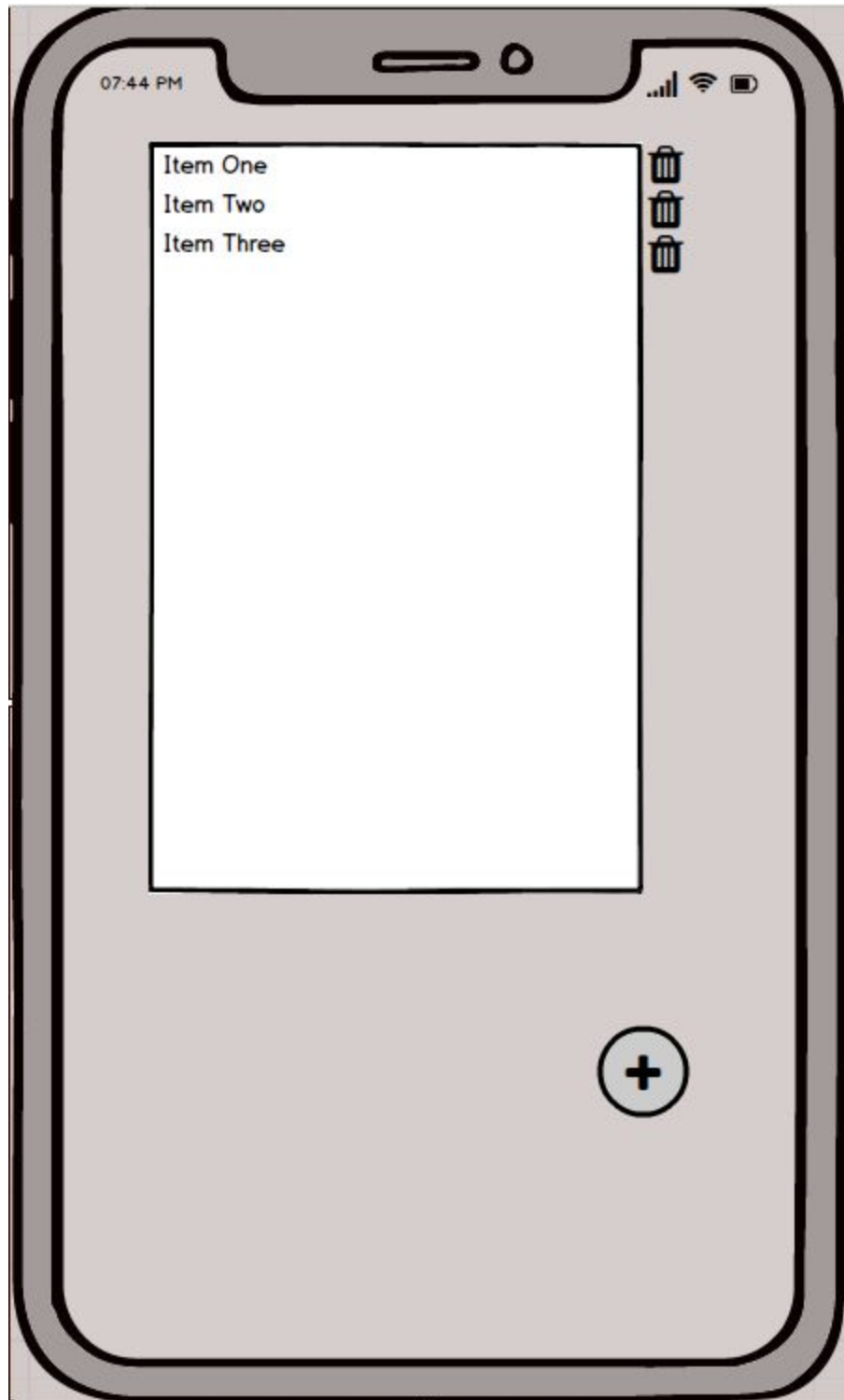
Device ID: 🔍

Product:

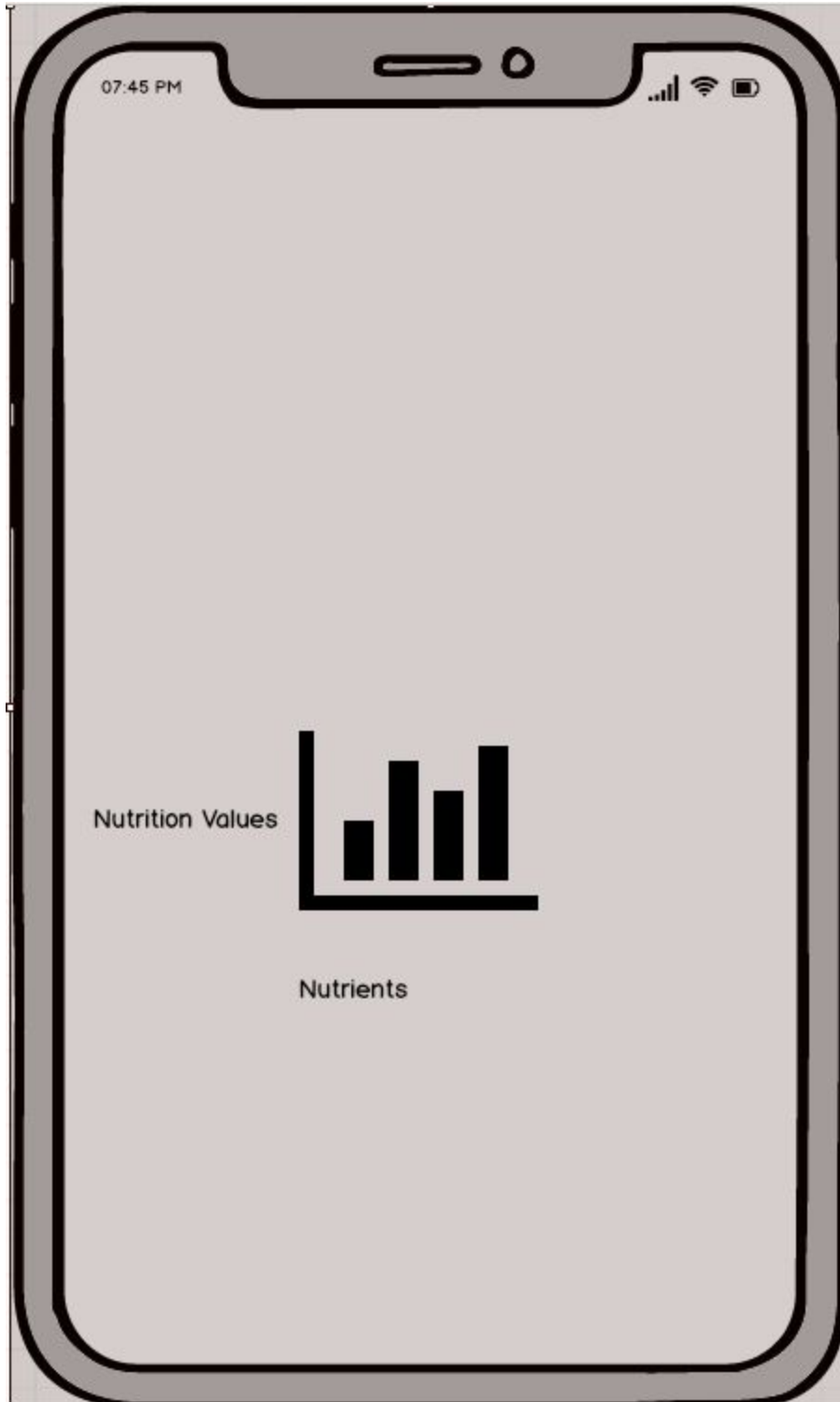
Threshold:

SAVE

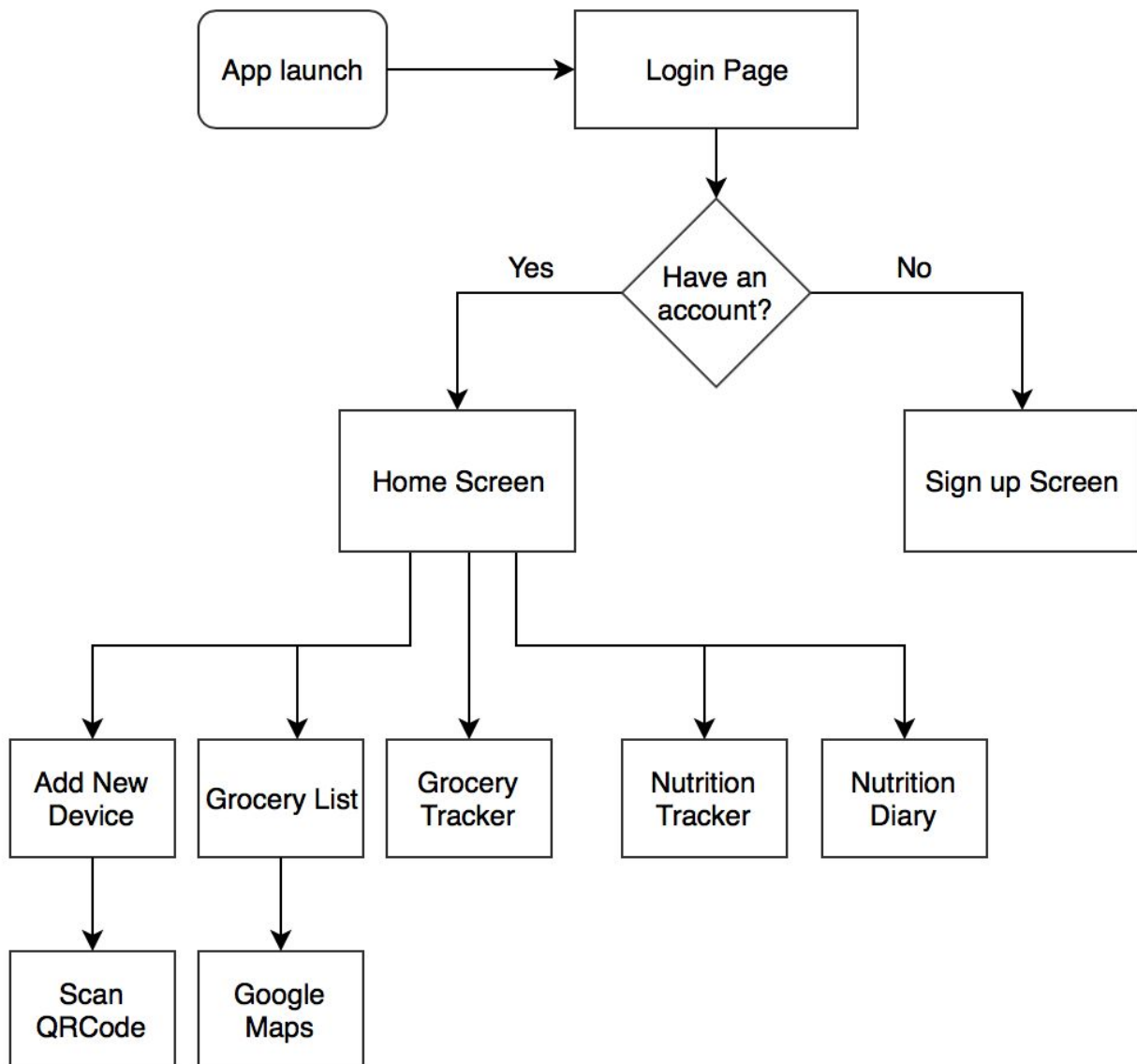
Todo List :



Nutrition Diary :



2.5 System Workflow

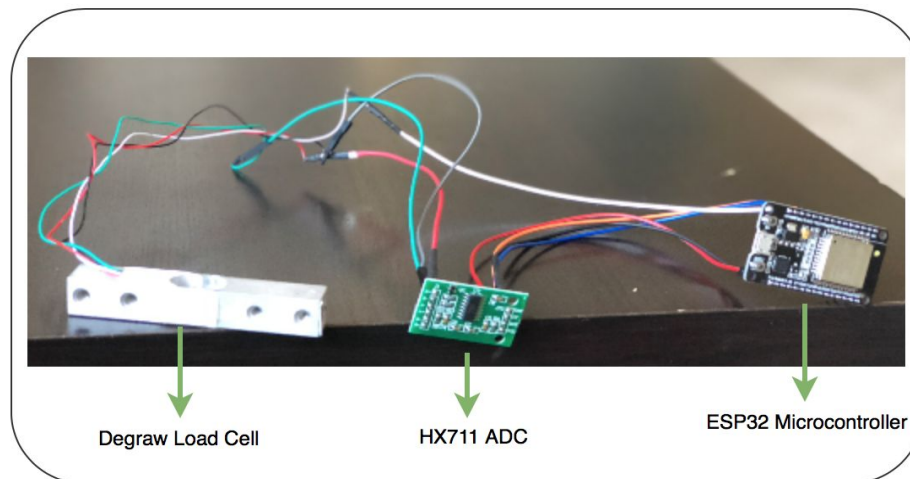


3. System Implementation

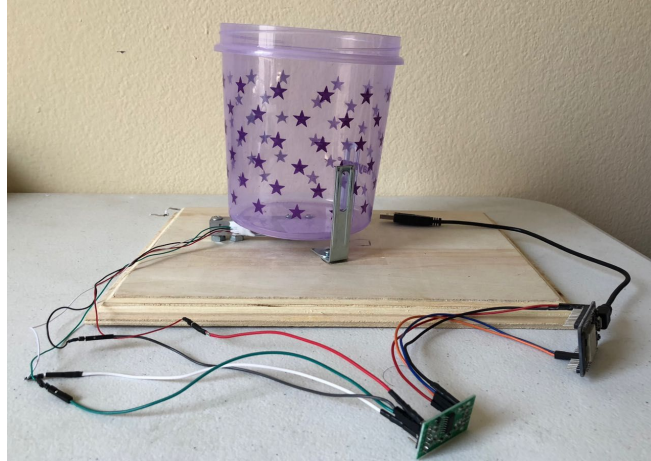
3.1 System Hardware Implementation Summary

3.1.1 IoT Component architecture

IoT is the new normal today. IoT is making significant growth as the application of IoT is everywhere in every field. Agriculture, home automation, automobile, retail industry are some of the fields where IoT is extensively used. The power of IoT is that it can transform any dumb device into an intelligent one to receive and send real-time and meaningful data. In our project, IoT plays an important role to send the weight data of the container to the AWS cloud. This information is further processed and analyzed to show some useful results on the mobile application MissingScoop.



Hardware setup



IoT Set up

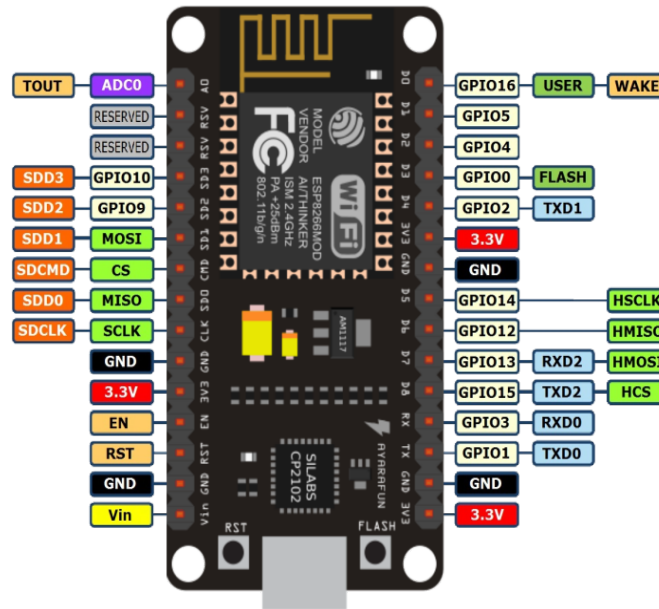
IoT set up normally consists of a system of devices(or things) connected to internet with IP addresses assigned to them to transfer or receive real time data. IoT set up in our project consists of the below components:

- Degraw Load cell
- HX711 ADC
- ESP32 Microcontroller
- MQTT client
- AWS IoT core
- AWS DynamoDB

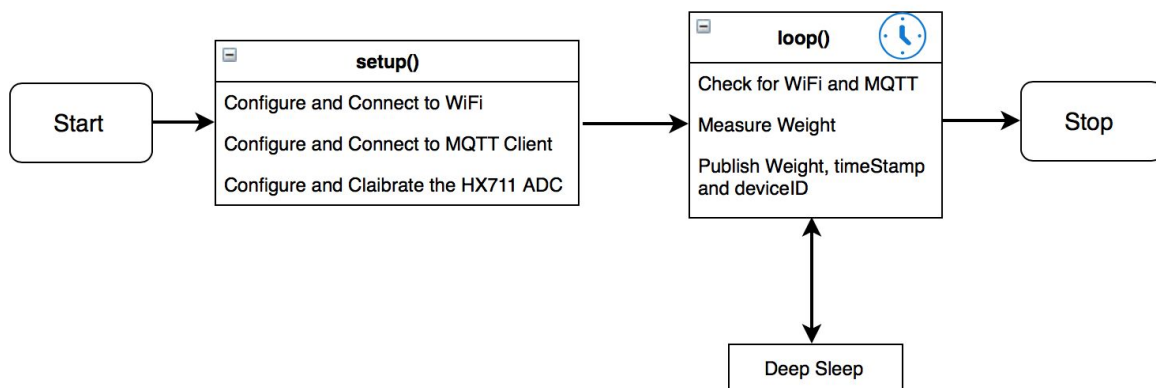
Degraw Load cell: Degraw Load cell weight sensor is a stress gauge which is used to measure the weight of the product that is placed inside the container. The load cell weight sensor measures the gauge as voltage.

HX711 ADC: In order to convert the voltage into measurable digital value we use HX711 analog to digital converter.

ESP32 Microcontroller: The microcontroller connects the load cell and HX711 to transmit the weight data with AWS IoT core. It provides power to HX711 and weight sensor. The data transfer from load cell to AWS IoT is achieved by embedding the C code inside the microcontroller. This code built using Arduino.



ESP32 Microcontroller Pin out diagram



Hardware embedded Program Flow

MQTT Client: MQTT client is the lightweight messaging protocol used to establish a connection between the IoT thing and AWS cloud. In our project, MQTT client is used to fetch the weight data from the load cell and push it to the cloud for further processing of data.

AWS IoT core: AWS IoT core allows the an easy and secure way for the devices to connect and communicate with the cloud applications. AWS IoT core is the crux of the IoT setup in our project, which subscribes to the data published by the weight sensor. AWS IoT helps to route this weight data to a DynamoDB table, which is our end point.

AWS DynamoDB: DynamoDB is a NoSQL database service provided by AWS. In our project, the weight data sent from the load cell weight sensor is routed to AWS IoT core. The rule engine

in AWS IoT is configured to receive data into DynamoDB as endpoint. DynamoDB consists of a table 'IoTData' that has all the weight data published by the device.

3.1.2 Connectivity Design

1. Make circuit connections from HX711 ADC and ESP32 microcontroller.
2. The next step is to embed the C code into microcontroller using Arduino. In order to do this download the Arduino software from the below link:

<https://www.arduino.cc/en/Main/Software>

3. In order to install Arduino to Mac follow the below steps:
Run the below commands.

```
mkdir -p ~/Documents/Arduino/hardware/espressif && \  
cd ~/Documents/Arduino/hardware/espressif && \  
git clone https://github.com/espressif/arduino-esp32.git esp32 && \  
cd esp32 && \  
git submodule update --init --recursive && \  
cd tools && \  
python get.py
```

4. Once installed, go to Arduino -> Tools ->Board and choose ESP32 Dev module since we are using ESP32 as the microcontroller in our setup.
5. Go to Arduino -> Tools ->Port->Slab_USBtoUART to configure the USB port.
6. Once Arduino is installed successfully using the above steps you can find some sample code kits to start off with some basics. Test a sample code on Arduino using the below steps before starting with the actual weight data:
 - a. Click the Up Arrow(Open). Go to BASICSà Blink
 - b. Add code in LED_BUILTIN = 2;
 - c. Click on Compile to compile the code.
 - d. Click on Upload to compile and update the code on to the microcontroller board. Now check if the LED on the ESP32 microcontroller is blinking with blue light. Now test code is working fine.
7. Copy the libraries WiFi, MQTT client, HX711 to the Arduino folder and then close and reopen Arduino.

```

#include "HX711.h"
#include "soc/rtc.h"
#include <math.h>
#include <TimeLib.h>
#include <WiFiClientSecure.h>
#include <MQTTClient.h>
#include <WiFiUdp.h>
// #include <NTPClient.h>

HX711 scale;
WiFiClientSecure net;
MQTTClient client;

```

- **WiFiClientSource Library:** This library helps to connect the device to WiFi (SSID and password) with which the data can be uploaded to the cloud.
- **MQTT library:** The MQTT client lets you to publish the weight data to the cloud.
Eg: client.publish() method publishes the weight data to IoT core.
- **HX711 library:** This library helps in calibrating and fetching the weight data from HX711 ADC. Eg: Scale.getunits() method is used to configure the weight scale

```

void setup() {
  Serial.begin(115200);
  rtc_clk_cpu_freq_set(RTC_CPU_FREQ_80M);

  WiFi.begin(ssid, pass);
  net.setCACert(rootCABuff);
  net.setCertificate(certificateBuff);
  net.setPrivateKey(privateKeyBuff);

  client.begin(awsEndPoint, 8883, net);
  connect();

  scale.begin(26, 25);
}

```

8. Once the ESP32 microcontroller has the power supply, the embedded C code gets executed and starts the setup and makes an attempt to connect to WiFi. Once the WiFi connection is successful, we try connecting to the MQTT client using the X.509 certificates provided by AWS IoT core, AWS endpoint, port 8883 and the publish topic. On the other hand of the microcontroller to fetch the data from HX711 ADC we set up the scale data to calibrate it to 0 and then start publishing this weight data along with device ID and timestamp where the AWS IoT core will subscribe to the same topic.

```

void loop() {

    if (!client.connected()) {
        connect();
    }

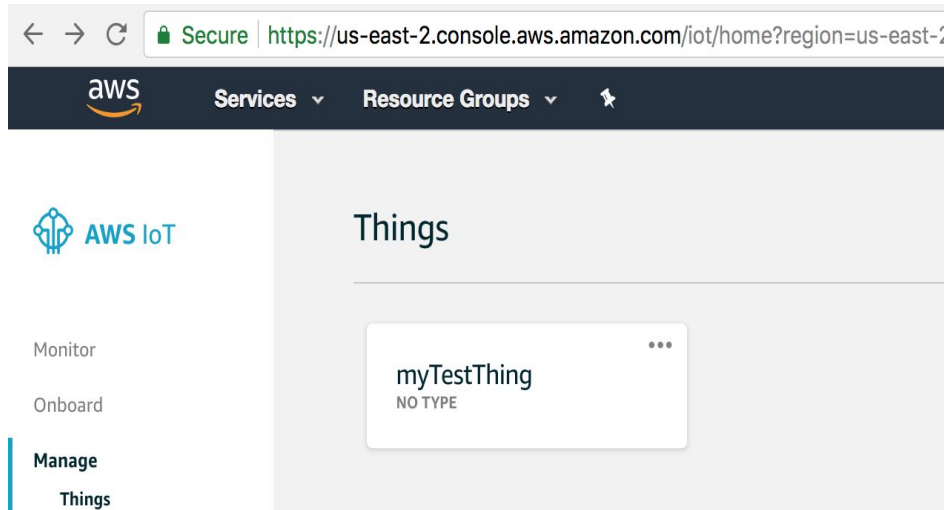
    client.loop();
    delay(100);

    if (millis() - lastMillis > 500) {
        lastMillis = millis();
        weight = roundNo(scale.get_units(20));
        if(weight<0) weight =0;
        time_t t = now();
        sprintf(msg, 130, "{\"timestamp\": \"%ld\", \"weight\": \"%d\", \"deviceId\": \"0001\", \"id\": \"%ld&d\"}", t, weight, t, 01);
        Serial.print("Publish message: ");
        Serial.println(msg);
        client.publish(publishTopic, msg);
    }
}

```

9. There some configurations that has to be done on the AWS IoT core using the AWS console. In order to subscribe to the weight data the following needs to be done on the AWS IoT core:

a. Login into AWS Console and create a new thing in AWS IoT by name 'myTestThing'.



b. Once the thing is created go to interact and get the topic name to publish and Subscribe with AWS MQTT client. Note this topic to add it the embedded C Code in order to publish the message

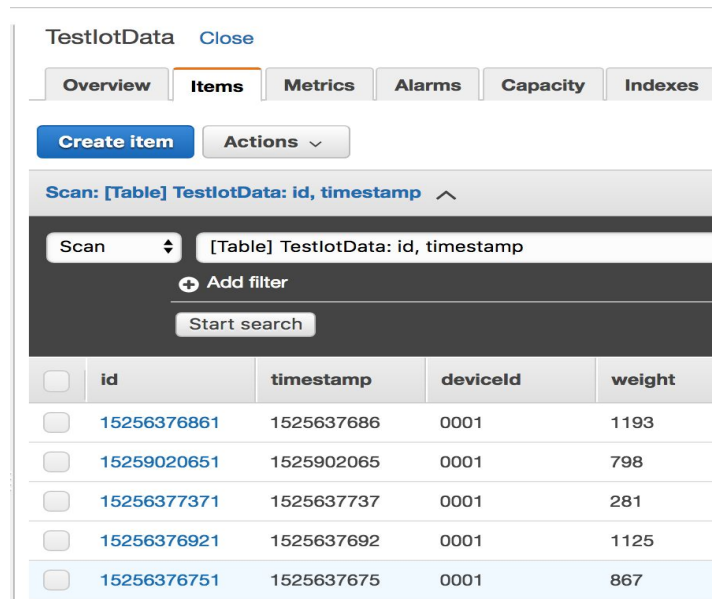
c. Generate certificates for the new thing on Security tab. There are three certificates that needs to be generated- the certificate for the thing, Root CA certificate and the private key certificate.

d. Create a policy to publish and subscribe to all clients and attach this policy to the certificate.

e. Verify the certificates and policies and add them to the embedded C code.

10. Create a DynamoDB rule in AWS IoT in the Act section and specify the table name and add required roles to perform a write to the table.

11. Once the data starts publishing to AWS IoT core the DynamoDB table will have the streaming weight data as shown below.



TestlotData Close

Overview **Items** Metrics Alarms Capacity Indexes

Create item Actions

Scan: [Table] TestlotData: id, timestamp

Scan [Table] TestlotData: id, timestamp

+ Add filter

Start search

	id	timestamp	deviceId	weight
<input type="checkbox"/>	15256376861	1525637686	0001	1193
<input type="checkbox"/>	15259020651	1525902065	0001	798
<input type="checkbox"/>	15256377371	1525637737	0001	281
<input type="checkbox"/>	15256376921	1525637692	0001	1125
<input type="checkbox"/>	15256376751	1525637675	0001	867

Datasheets:

ESP32 Microcontroller:

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

HX711: https://www.mouser.com/ds/2/813/hx711_english-1022875.pdf

Degraw Load cell: <https://www.robotshop.com/media/files/pdf/datasheet-3133.pdf>

3.2 Mobile and Cloud Technologies Description

The entire project is designed on enterprise integration patterns with appropriate cloud services invoked via RESTful APIs. The RESTful APIs are invoked using Volley library. The REST APIs are designed such that it performs all the CRUD operations on Dynamodb.

In the following sections we explain all the integrations, working model, code snippets of each of the feature implemented.

Registration, Login and Logout:

Firebase Authentication is used for storing and authorizing users to access the application. The firebase dependency has to be added in the project gradle script as below.

```
compile 'com.google.firebase:firebase-core:12.0.0'
compile 'com.google.firebase:firebase-database:12.0.0'
```



```
compile 'com.google.firebase:firebase-auth:12.0.0'
```

App users can register by signing up with a username and password. Username is accepted in email format. When user registers, the user information is saved in Firebase authentication component.

```
public void registerWithEmailAndPassword(String email, String password, final
SignUpActivity activity, final AuthenticationListener authenticationListener) {
    firebaseAuth.createUserWithEmailAndPassword(email,
password).addOnCompleteListener(activity, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                authenticationListener.onSuccess("Login Success!!");
            } else {
                authenticationListener.onFailure("Login Failed!!");
            }
        }
    });
}
```

Once registered, users can login to the app by previously created credentials. Firebase is used to maintain the current user session and the session times out when user logs out.

Logout is also handled by invoking firebase logout function which removes user from session and invalidate user session.

```
public void signInWithEmailAndPassword(String email, String password, final
MainActivity activity, final AuthenticationListener authenticationListener) {
    firebaseAuth.signInWithEmailAndPassword(email,
password).addOnCompleteListener(activity, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                authenticationListener.onSuccess("Login Success!!");
            } else {
                authenticationListener.onFailure("Login Failed!!");
            }
        }
    });
}

public void logout() {
    firebaseAuth.signOut();
}
```

Assign Device and Map Product:

To identify which device consists of what product for efficient tracking mechanism, we allow the user to map the device and product name. The device qrcode is scanned and then associated with a product name. The qrcode is detected using google's visions API.

To operate on google visions, include the below dependency in gradle build

```
compile 'com.google.android.gms:play-services-vision:12.0.0'
```

Below is the code snippets for detecting qrcode

```
public void scanBarcode() {
    cameraPreview = findViewById(R.id.surfaceView);

    barcodeDetector = new
BarcodeDetector.Builder(this).setBarcodeFormats(Barcode.QR_CODE).build();
    cameraSource = new CameraSource.Builder(this,
barcodeDetector).setRequestedPreviewSize(640,
480).setFacing(CameraSource.CAMERA_FACING_BACK).build();

    cameraPreview.getHolder().addCallback(new SurfaceHolder.Callback() {
        @Override
        public void surfaceCreated(SurfaceHolder surfaceHolder) {
            if (ActivityCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(QRCodeScanActivity.this, new
String[]{Manifest.permission.CAMERA}, requestCameraPermissionId);
                return;
            }
            try {
                cameraSource.start(cameraPreview.getHolder());
            } catch (IOException e) {
                Log.e("QRCodeScanActivity", e.getMessage());
            }
        }

        @Override
        public void surfaceChanged(SurfaceHolder surfaceHolder, int i, int i1, int
i2) {

        }

        @Override
        public void surfaceDestroyed(SurfaceHolder surfaceHolder) {
            cameraSource.stop();
        }
    });

    barcodeDetector.setProcessor(new Detector.Processor<Barcode>() {
        @Override
        public void release() {

        }

        @Override
        public void receiveDetections(Detector.Detections<Barcode> detections) {
```

```

        SparseArray<Barcode> qrCodes = detections.getDetectedItems();
        if (qrCodes.size() != 0 && !alreadyScanned) {
            alreadyScanned = true;

            String qrCodeValue = qrCodes.valueAt(0).displayValue;
            Log.d("QRCodeScanActivity", qrCodeValue);

            setActivityResult(qrCodeValue);
        }
    });
}

```

Once the user scans QRCode and chooses the product to be mapped, the mapping is saved in dynamodb. Rest APIs are invoked to save this data on dynamodb.

```

@Override
public void save(DeviceProductMapping deviceProductMapping) {
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();
    DynamoDBMapper mapper = new DynamoDBMapper(dynamoDB);
    mapper.save(deviceProductMapping);
}

```

Grocery Tracker - Product List:

User can keep a track of the weight of the products that they have registered by using the Grocery Tracking functionality. The list of all the registered products is obtained by querying the DynamoDB table to fetch all the products that user has registered in the app.

```

@Override
public List<DeviceProductMapping> getDeviceProductMappingByUserName(String
userName) {
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();
    DynamoDBMapper mapper = new DynamoDBMapper(dynamoDB);

    Map<String, AttributeValue> values = new HashMap<String, AttributeValue>();
    values.put(":userName", new AttributeValue().withS(userName));
    DynamoDBScanExpression scanExpression = new
DynamoDBScanExpression()
        .withFilterExpression("userName =
:userName").withExpressionAttributeValues(values);

    List<DeviceProductMapping> deviceProductMappingList =

```

```

mapper.scan(DeviceProductMapping.class, scanExpression);
        return deviceProductMappingList;
    }

```

The resulting list is shown on Android UI by using the Card View and Recycler View. To use on cardview and recyclerview, include the below dependencies in gradle build.

```

compile 'com.android.support:cardview-v7:26.+'
compile 'com.android.support:recyclerview-v7:26.+'

```

Below is the code snippet on Android to call the backend API and display the product list in a recyclerview.

```

public void getDeviceProductMapping() {
    UserInfo user = authenticationHandler.getCurrentUser();
    String uri = "/fetch/device/product?userName=" + user.getEmail();
    restApiClient.executeGetAPI(getApplicationContext(), uri, new VolleyAPICallback() {
        @Override
        public void onSuccess(JSONObject jsonResponse) {
            response = gson.fromJson(jsonResponse.toString(), DeviceProductListResponse.class);
            Log.i("ListProductActivity", response.toString());

            List<DeviceProductMappingResponse> responseList = response.getResponse();
            for (DeviceProductMappingResponse record : responseList) {
                labelsList.add(record.getLabel());
                labelsDeviceMap.put(record.getDeviceId(), record.getLabel());
            }
            Log.i("Product List", labelsList.toString());

            final RVAdapter adapter = new RVAdapter(labelsList);
            rv.setAdapter(adapter);
        }
    });
}

```

```

@Override
public ViewHolder onCreateViewHolder(ViewGroup viewGroup, int i) {
    View v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.content_main, viewGroup, false);
    context = viewGroup.getContext();
    ViewHolder pvh = new ViewHolder(v);
    return pvh;
}

@Override
public void onBindViewHolder(ViewHolder viewHolder, int i) {
    viewHolder.product.setText(labels.get(i));
}

```

Grocery Tracker - Product Details:

The user can explore the details of the every product he has registered in the app. The details include current weight of the device, consumption rate and estimated completion days of the product. The weight of the product is continuously tracked by the IoT device and gets updated in the DynamoDB table. The weight from the DynamoDB is fetched by pulling the records of that device that user is interested in and ordering the data in descending order of timestamp.

@Override

```
public DeviceWeightResponse getDeviceWeightByDeviceId(String deviceId) {

    DeviceWeightResponse finalResponse = new DeviceWeightResponse();
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();
    Map<String, String> expressionAttributesNames = new HashMap<>();
    expressionAttributesNames.put("#deviceId", "deviceId");

    Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
    expressionAttributeValues.put(":deviceIdValue", new AttributeValue().withS(deviceId));

    QueryRequest queryRequest = new QueryRequest().withTableName("TestIotData")
        .withIndexName("deviceId-timestamp-index")
        .withKeyConditionExpression("#deviceId = :deviceIdValue")
        .withExpressionAttributeNames(expressionAttributesNames)

.withExpressionAttributeValues(expressionAttributeValues).withScanIndexForward(false);

    QueryResult queryResult = dynamoDB.query(queryRequest);
    List<Map<String, AttributeValue>> results = queryResult.getItems();
```

The top entry in the records fetched gives the current weight of the product in grams.

```
double currentWeight = Double.parseDouble((results.get(0)).get("weight").getS());
finalResponse.setCurrentWeight(currentWeight);
```

The consumption rate will be the total consumption of the product divided by the number of days since the refill. This will be measured as grams per day.

```
for (i = 0; i < results.size() - 1; i++) {
    entry = Double.parseDouble((results.get(i)).get("weight").getS());
    preentry = Double.parseDouble((results.get(i + 1)).get("weight").getS());
    if (entry < 0)
        entry = 0;
    if (preentry < 0)
        preentry = 0;
    temp = entry - preentry;
    if (temp > 10)
        break;
```

```

        if (temp > 0)
            continue;
        summ += preventry - entry;
    }
    if (summ > 0) {
        int lastdate = Integer.parseInt((results.get(0)).get("timestamp").getS());
        int refilledDate = Integer.parseInt((results.get(i)).get("timestamp").getS());
        int divisor = 1;
        if (refilledDate - lastdate < 86400)
            divisor = 1;
        else
            divisor = (refilledDate - lastdate) / 86400;
        double consumptionrate = summ / divisor;
        finalResponse.setConsumptionRate(consumptionrate);
    }

```

The estimated completion days will be the quotient of the current weight and the consumption rate. This will be measured in days.

```

double estimatedCompletionDays = currentWeight / consumptionrate;
finalResponse.setEstimatedCompletion(estimatedCompletionDays);

```

The calculated final response is sent to Android UI by invoking a volley API and displayed to user. Based on the weight of the product the image for the container is displayed on UI. The code snippet on the Android end to call the API and set UI elements is show below:

```

public void getDeviceWeight(final String device) {
    String uri = "/fetch/device/weight?deviceId=" + device;
    RestClient.executeGetAPI(getApplicationContext(), uri, new VolleyAPICallback() {
        @Override
        public void onSuccess(JSONObject jsonResponse) {
            response = gson.fromJson(jsonResponse.toString(), DeviceWeightResponse.class);
            Log.i("ProductDetailsActivity", response.toString());

            weight = response.getCurrentWeight();
            consumptionRate = response.getConsumptionRate();
            estimatedCompletion = response.getEstimatedCompletion();

            String sourceString1 = "<b>" + "Device " + device + "</b> ";
            deviceTextView.setText(Html.fromHtml(sourceString1));

            String sourceString2 = "Remaining " + "<b>" + item + ": " + weight + "</b> " + " grams";
            currentWeight.setText(Html.fromHtml(sourceString2));

            String sourceString3 = "Consumption Rate: " + "<b>" + consumptionRate + "</b> " + " grams per day";
            consumptionRateTextView.setText(Html.fromHtml(sourceString3));

            String sourceString4 = "Estimated Completion: " + "<b>" + estimatedCompletion + "</b> " + " day/s";
            estimatedCompletionTextView.setText(Html.fromHtml(sourceString4));

            if(weight<200)
                weightStatusImage.setImageResource(R.drawable.empty);
        }
    });
}

```

```

else if(weight>200 && weight<500)
    weightStatusImage.setImageResource(R.drawable.medium2);
else if(weight>500 && weight<700)
    weightStatusImage.setImageResource(R.drawable.medium1);
else if(weight>700)
    weightStatusImage.setImageResource(R.drawable.full);
}

```

Nutrition Charts:

User grocery consumption is continuously tracked using the IoT device data that is updated on Dynamodb. From this data, a total consumption of the product is evaluated. The total consumption computation is done by calculating the difference in weight state in IoT data. The weight captured at each point is ordered by timestamp and then the total consumption is evaluated by identifying if the difference is negative or positive. A positive difference in 2 states is considered a consumption as ordering is done by ascending timestamp. A negative difference is a refill.

Below is the dynamodb query for fetching records IoT table per device and ordered in ascending order of timestamp

```

@Override
public List<Double> getWeight(String deviceId, Long fromTimestamp, Long
toTimestamp, boolean sort) {
    List<Double> returnList = new ArrayList<>();
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();
    Map<String, String> expressionAttributesNames = new HashMap<>();
    expressionAttributesNames.put("#deviceId", "deviceId");
    expressionAttributesNames.put("#timestamp", "timestamp");

    Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
    expressionAttributeValues.put(":deviceIdValue", new
AttributeValue().withS(deviceId));
    expressionAttributeValues.put(":to", new
AttributeValue().withS(String.valueOf(toTimestamp)));
    expressionAttributeValues.put(":from", new
AttributeValue().withS(String.valueOf(fromTimestamp)));

    QueryRequest queryRequest = new
QueryRequest().withTableName("TestlotData")
                .withIndexName("deviceId-timestamp-index")
                .withKeyConditionExpression("#deviceId = :deviceIdValue and
#timestamp BETWEEN :from AND :to ")
                .withExpressionAttributeNames(expressionAttributesNames)
                .withExpressionAttributeValues(expressionAttributeValues).withScanIndexForward(sort);

```

```

        QueryResult queryResult = dynamoDB.query(queryRequest);
        List<Map<String, AttributeValue>> results = queryResult.getItems();

        if (CollectionUtils.isNullOrEmpty(results)) {
            return null;
        }
        for (Map<String, AttributeValue> result : results) {
            returnList.add(Double.parseDouble(result.get("weight").getS()));
        }
        return returnList;
    }

```

Below is the code snippet for calculating total consumption.

```

@Override
public Double calculateTotalConsumption(String deviceId) {
    Double totalConsumption = new Double(0);
    Calendar currentDate = Calendar.getInstance();

    currentDate.set(Calendar.HOUR_OF_DAY, 23);
    currentDate.set(Calendar.MINUTE, 55);
    currentDate.set(Calendar.SECOND, 0);
    currentDate.set(Calendar.MILLISECOND, 0);

    Long toTimestamp = currentDate.getTimeInMillis();

    currentDate.set(Calendar.HOUR_OF_DAY, 0);
    currentDate.set(Calendar.MINUTE, 0);
    currentDate.set(Calendar.SECOND, 1);
    currentDate.set(Calendar.MILLISECOND, 0);

    Long fromTimestamp = currentDate.getTimeInMillis();

    List<Double> weights = deviceWeightDao.getWeight(deviceId,
fromTimestamp, toTimestamp, true);
    if (CollectionUtils.isNullOrEmpty(weights)) {
        return totalConsumption;
    }

    for (int index = 0; index < weights.size() - 1; index++) {
        if (weights.get(index) > weights.get(index + 1)) {
            totalConsumption = totalConsumption + (weights.get(index) -
weights.get(index + 1));

```



```

        }
    }
    return totalConsumption;
}

```

Once the consumption is evaluated, from the device product mapping the product nutrient values are fetched and a nutrition chart is drawn for users to track their nutrition consumed. Open source library AChartEngine is used to draw bar graphs. Include the below dependency in gradle.

```
compile group: 'org.achartengine', name: 'achartengine', version: '1.2.0'
```

Bar charts using AChartEngine can be drawn using below code snippets

```

private void openChart(UserNutritionResponse response) {

    if (null == response) {
        return;
    }

    double[] nutritionValues = new double[7];
    nutritionValues[0] = response.getCarbohydrate();
    nutritionValues[1] = response.getProtein();
    nutritionValues[2] = response.getFat();
    nutritionValues[3] = response.getFiber();
    nutritionValues[4] = response.getSugar();
    nutritionValues[5] = response.getSodium();

    XYSeries nutritionValueSeries = new XYSeries("Nutrition Values");
    for (int i = 0; i < mNutritions.length; i++) {
        nutritionValueSeries.add(i, nutritionValues[i]);
    }

    XYMultipleSeriesDataset dataset = new XYMultipleSeriesDataset();
    dataset.addSeries(nutritionValueSeries);

    XYSeriesRenderer nutritionValueRenderer = new XYSeriesRenderer();
    nutritionValueRenderer.setColor(Color.rgb(130, 130, 230));
    nutritionValueRenderer.setFillPoints(true);
    nutritionValueRenderer.setLineWidth(2);
    nutritionValueRenderer.setDisplayChartValues(true);

    XYMultipleSeriesRenderer multiRenderer = new XYMultipleSeriesRenderer();
    multiRenderer.setXLabels(0);
    multiRenderer.setXTitle("Nutrients");
    multiRenderer.setYTitle("Nutrition Values");
    multiRenderer.setLabelsColor(Color.BLACK);

    for (int i = 0; i < mNutritions.length; i++) {
        multiRenderer.addXTextLabel(i, mNutritions[i]);
    }

    multiRenderer.addSeriesRenderer(nutritionValueRenderer);
}

```

```

        LinearLayout chartContainer = (LinearLayout) findViewById(R.id.chart2);
        chartContainer.removeAllViews();
        View mChart = ChartFactory.getBarChartView(getBaseContext(), dataset,
        multiRenderer, BarChart.Type.DEFAULT);
        chartContainer.addView(mChart);
    }

```

Push Notifications:

When user tracked grocery goes below threshold, user has to be notified. Google cloud messaging is used to enable push notifications.

When a user login to the app, the device id is captured and saved in dynamodb. This device id is later used for sending notification to a specific user

```

FirebaseInstanceId.getInstance().getToken()

```

A cron job monitors IoT data for weights going below the threshold. When weight reaches below threshold, user is notified on the registered device.

To enable push notification add the below dependency in gradle

```

compile 'com.google.firebase:firebase-messaging:12.0.0'

```

Call Google's rest APIs to send notification.

```

@Async
public CompletableFuture<String> send(HttpEntity<String> entity) {

    RestTemplate restTemplate = new RestTemplate();

    ArrayList<ClientHttpRequestInterceptor> interceptors = new ArrayList<>();
        interceptors.add(new HeaderRequestInterceptor("Authorization", "key=" +
        FIREBASE_SERVER_KEY));
        interceptors.add(new HeaderRequestInterceptor("Content-Type", "application/json"));
        restTemplate.setInterceptors(interceptors);

                                String      firebaseResponse      =
    restTemplate.postForObject("https://fcm.googleapis.com/fcm/send", entity, String.class);

    return CompletableFuture.completedFuture(firebaseResponse);
}

```

Extend `FirebaseMessagingService` class to receive push notification message and display it to users.

```
public class PushNotificationService extends FirebaseMessagingService {
    public PushNotificationService() {
    }

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        super.onMessageReceived(remoteMessage);
        Log.d("PushNotificationService", "From: " + remoteMessage.getFrom());
        Log.d("PushNotificationService", "Notification Message Body: " +
remoteMessage.getNotification().getBody());

        sendNotification(remoteMessage);
    }
}
```

Add/Remove Groceries:

To allow users to add groceries to the todo list , we get the user input and add to list view as well as adding record to dynamodb.

The following is the snippets for adding grocery to the todo list

```
public void addGrocery() {
    UserInfo user = authenticationHandler.getCurrentUser();
    GroceryListRequest request = new GroceryListRequest();
    request.setGrocery(groceryEditText.getText().toString());
    request.setUserName(user.getEmail());
    try {
        JSONObject jsonObject = new JSONObject(gson.toJson(request));
        restApiClient.executePostAPI(getApplicationContext(), "/add/grocery",
            jsonObject, new VolleyAPICallback() {
                @Override
                public void onSuccess(JSONObject jsonResponse) {
                    GrocerListResponse response = gson.fromJson(jsonResponse.toString(),
                        GrocerListResponse.class);
                    Log.i("GroceryListActivity", response.toString());
                    addToListView(response.getGroceryList());
                }

                @Override
                public void onError(String message) {

                    Log.i("GroceryListActivity", message);
                }
            });
    } catch (JSONException e) {
        Log.e("GroceryListActivity", e.getMessage());
    }
}
```

The following is snippets for adding groceries to list view. To ensure that the page load as empty list even if there are no data in grocery list DynamoDB table, a checkpoint was added to initialize the list to empty list whenever the condition met .

```
private void addToListView(List<String> groceryList) {  
    if (groceryList == null) {  
        groceryList = new ArrayList<String>();  
    }  
    if (adapter == null) {  
        adapter = new ArrayAdapter<String>(GroceryListActivity.this, R.layout.row,  
            R.id.grocery_name, groceryList);  
        lvItems.setAdapter(adapter);  
    } else {  
        adapter.clear();  
        adapter.addAll(groceryList);  
        adapter.notifyDataSetChanged();  
    }  
}
```

Below is the snippet if code to invoke rest API to save grocery into DynamoDB

```
@Override  
public void save(GroceryList groceryList) {  
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();  
    DynamoDBMapper mapper = new DynamoDBMapper(dynamoDB);  
    GroceryList dbGrocery =  
getGroceriesByUserName(groceryList.getUserName());  
  
    if (null == dbGrocery) {  
        dbGrocery = groceryList;  
    } else {  
        List<String> dbGroceryList = dbGrocery.getGrocery();  
        dbGroceryList.addAll(groceryList.getGrocery());  
        dbGrocery.setGrocery(dbGroceryList);  
    }  
    mapper.save(dbGrocery);  
}
```

To allow users to remove each grocery from their lists , the following snippets was used

```

public void removeGrocery(String grocery) {
    UserInfo user = authenticationHandler.getCurrentUser();
    GroceryListRequest request = new GroceryListRequest();
    request.setUserName(user.getEmail());
    request.setGrocery(grocery);
    try {
        JSONObject jsonObject = new JSONObject(gson.toJson(request));
        restApiClient.executePostAPI(getApplicationContext(), "/remove/grocery",
            jsonObject, new VolleyAPICallback() {
                @Override
                public void onSuccess(JSONObject jsonResponse) {
                    GrocerListResponse response = gson.fromJson(jsonResponse.toString(),
                        GrocerListResponse.class);
                    Log.i("GroceryListActivity", response.toString());
                    //loadGroceryList();
                    addToListView(response.getGroceryList());
                }

                @Override
                public void onError(String message) {
                    Log.i("GroceryListActivity", message);
                }
            });
    } catch (JSONException e) {
        Log.e("GroceryListActivity", e.getMessage());
    }
}

```

Below is the snippet of code to invoke rest API to remove grocery from DyamoDB

```

@Override
public GroceryList getGroceriesByUserName(String userName) {
    AmazonDynamoDB dynamoDB = dynamodbClient.getDynamoDB();
    DynamoDBMapper mapper = new DynamoDBMapper(dynamoDB);

    Map<String, AttributeValue> values = new HashMap<String, AttributeValue>();
    values.put(":userName", new AttributeValue().withS(userName));
    DynamoDBScanExpression scanExpression = new
DynamoDBScanExpression()
        .withFilterExpression("userName =
:userName").withExpressionAttributeValues(values);

    List<GroceryList> groceryList = mapper.scan(GroceryList.class,
scanExpression);
    if (!CollectionUtils.isNullOrEmpty(groceryList)) {
        return groceryList.get(0);
    }
    return null;
}

```

Google Maps - Show nearby Grocery Stores:

Google maps in grocery list page lets the user to look out for nearby grocery stores.

Add the below dependency in gradle for google play services:

```
compile 'com.google.android.gms:play-services-location:15.0.1'
```

Using the Google maps API Key and Google places API get the list of nearby grocery stores in the surrounding vicinity of 10000 meters from current User location.

```
StringBuilder googlePlacesUrl = new  
StringBuilder("https://maps.googleapis.com/maps/api/place/nearbysearch/json?");  
googlePlacesUrl.append("location=" + latitude + "," + longitude);  
googlePlacesUrl.append("&radius=" + proximityRadius);  
googlePlacesUrl.append("&type=" + nearbyPlace);  
googlePlacesUrl.append("&sensor=true");  
googlePlacesUrl.append("&key=" + "<YOUR_API_KEY>");  
Log.d("getUrl", googlePlacesUrl.toString());  
return (googlePlacesUrl.toString());
```

Create a google map activity and then locate these places on the google maps.

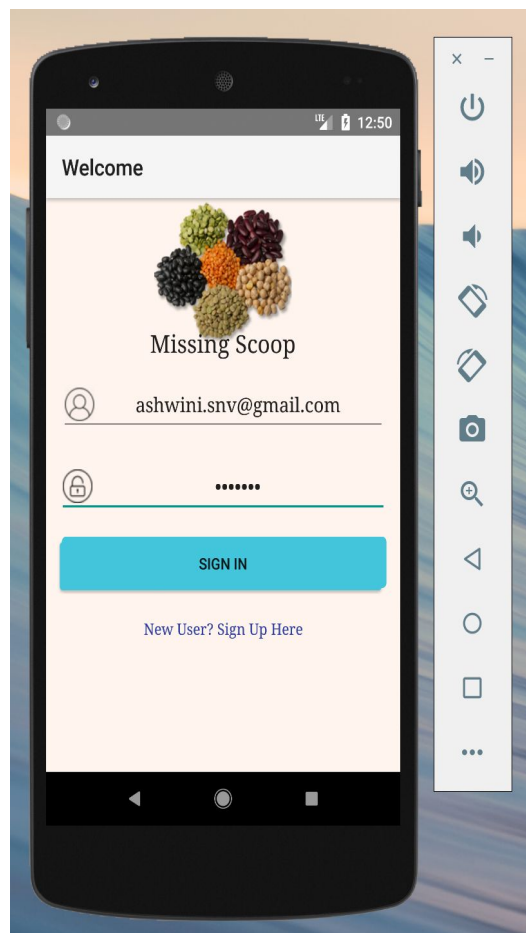
```
private void ShowNearbyPlaces(List<HashMap<String, String>> nearbyPlacesList) {  
    for (int i = 0; i < nearbyPlacesList.size(); i++) {  
        MarkerOptions markerOptions = new MarkerOptions();  
        HashMap<String, String> googlePlace = nearbyPlacesList.get(i);  
        double lat = Double.parseDouble(googlePlace.get("lat"));  
        double lng = Double.parseDouble(googlePlace.get("lng"));  
        String placeName = googlePlace.get("place_name");  
        String vicinity = googlePlace.get("vicinity");  
        LatLng latLng = new LatLng(lat, lng);  
        markerOptions.position(latLng);  
        markerOptions.title(placeName + " : " + vicinity);  
        gMap.addMarker(markerOptions);  
        markerOptions.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED));  
        //move map camera  
        gMap.moveCamera(CameraUpdateFactory.newLatLng(latLng));  
        gMap.animateCamera(CameraUpdateFactory.zoomTo(11));  
    }  
}
```

3.3 System Interface and Server Side Design

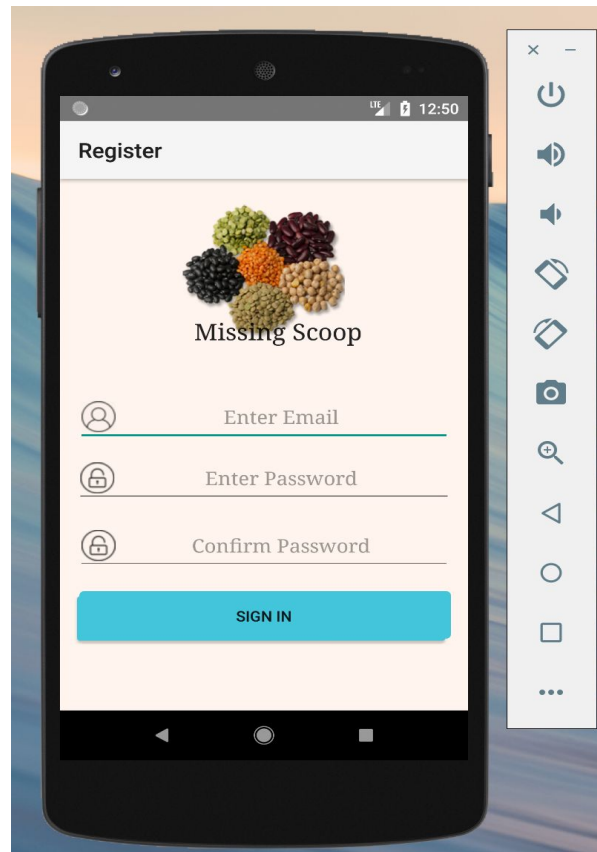
Server Side code is written using MVC design pattern. OOPs design principles are upheld. The entire backend server is exposed as RESTful APIs. The backend server is written using Java and hosted on AWS EC2 as Spring Boot Application. The APIs are invoked in Android using Volley library.

3.4 Client Side Design

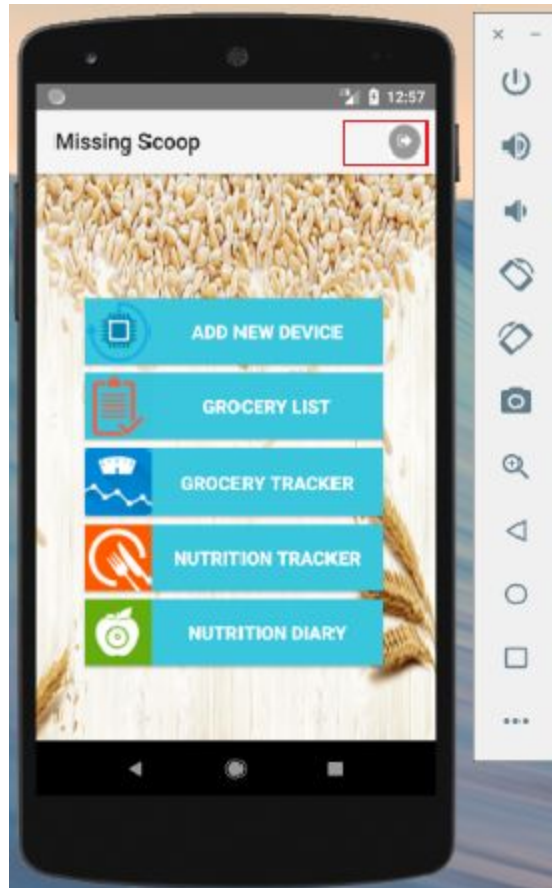
1. Sign-in Screen



2. Sign Up Screen

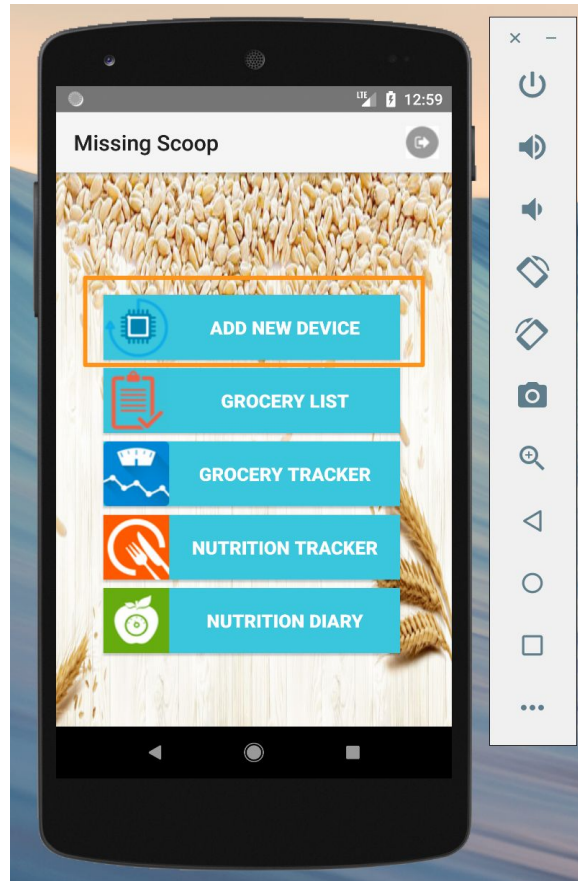


3. Home Screen with menu options and logout button

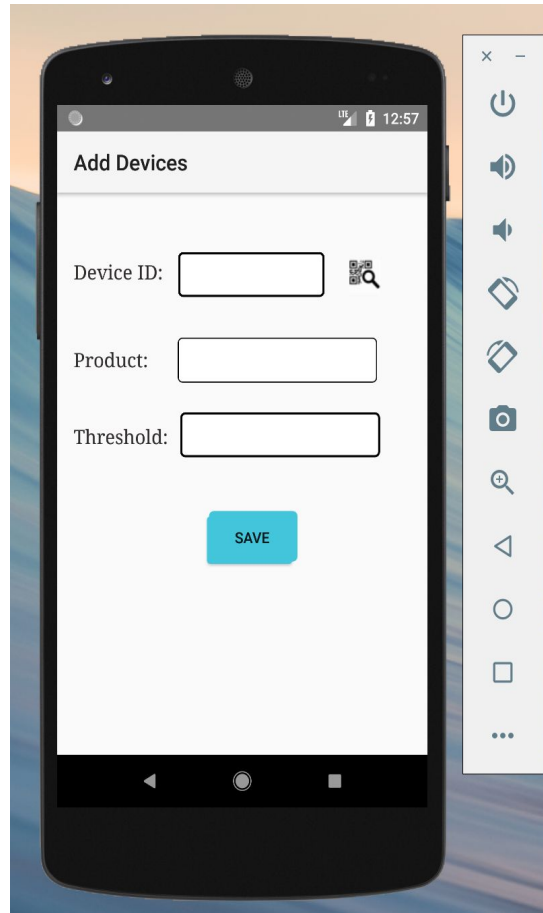


Add new device

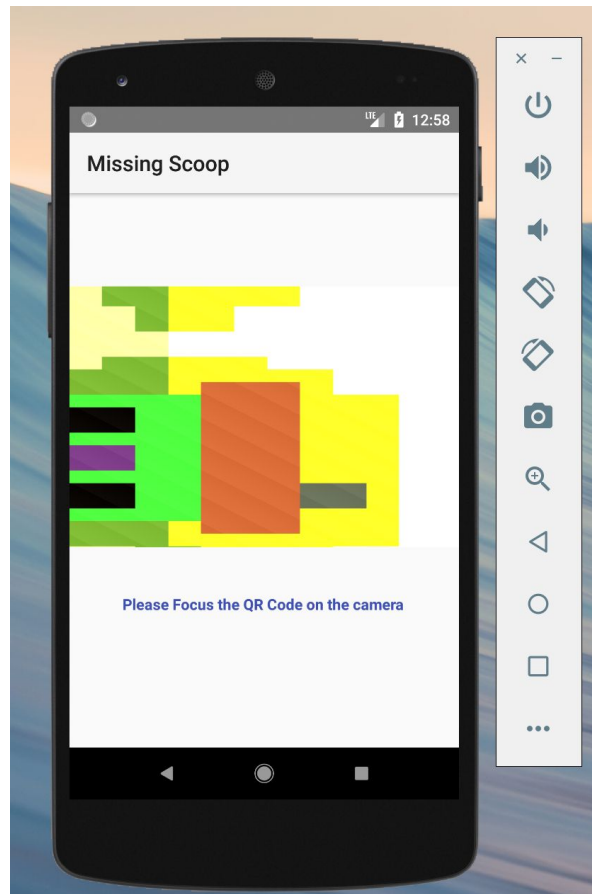
1. Select 'Add New Device' to add a new IoT device.



2. 'Add new device' screen consisting of QR code scanner, product and threshold input.

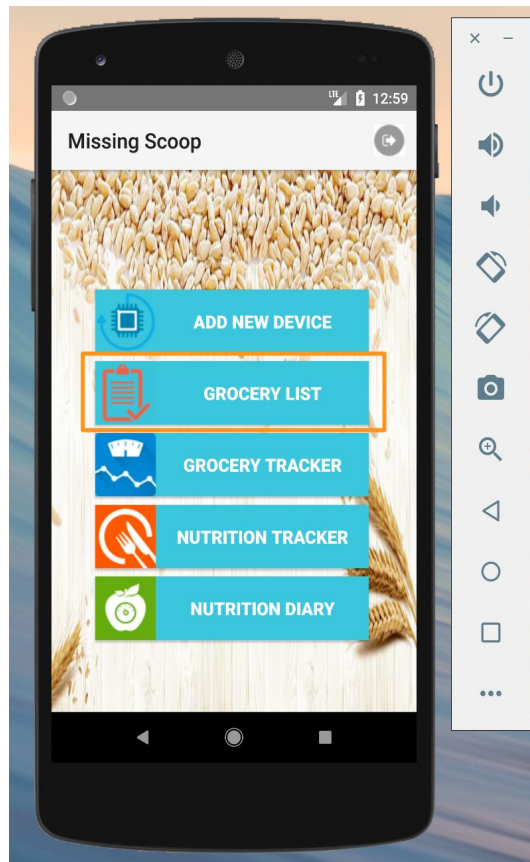


3. On click of QR Code Scanner, user can scan the QR code on the camera.

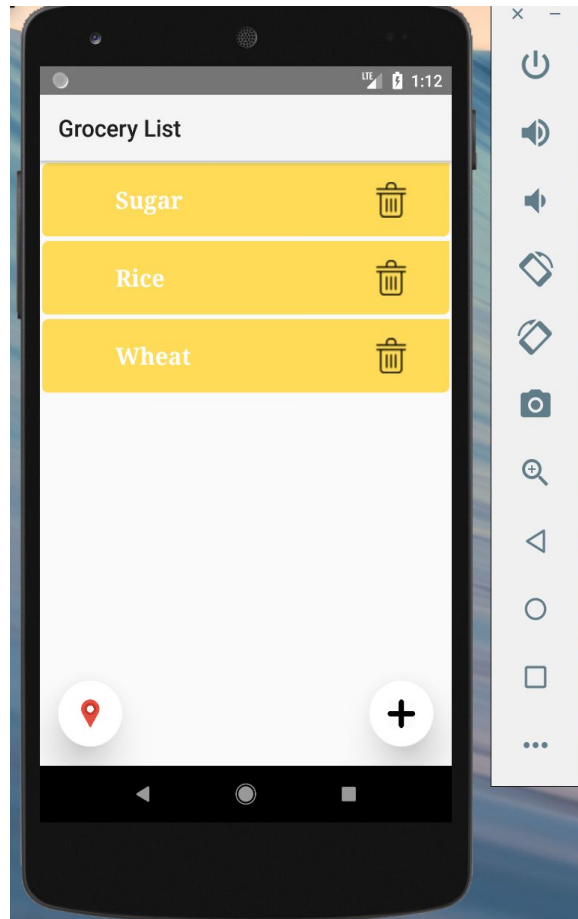


Grocery List

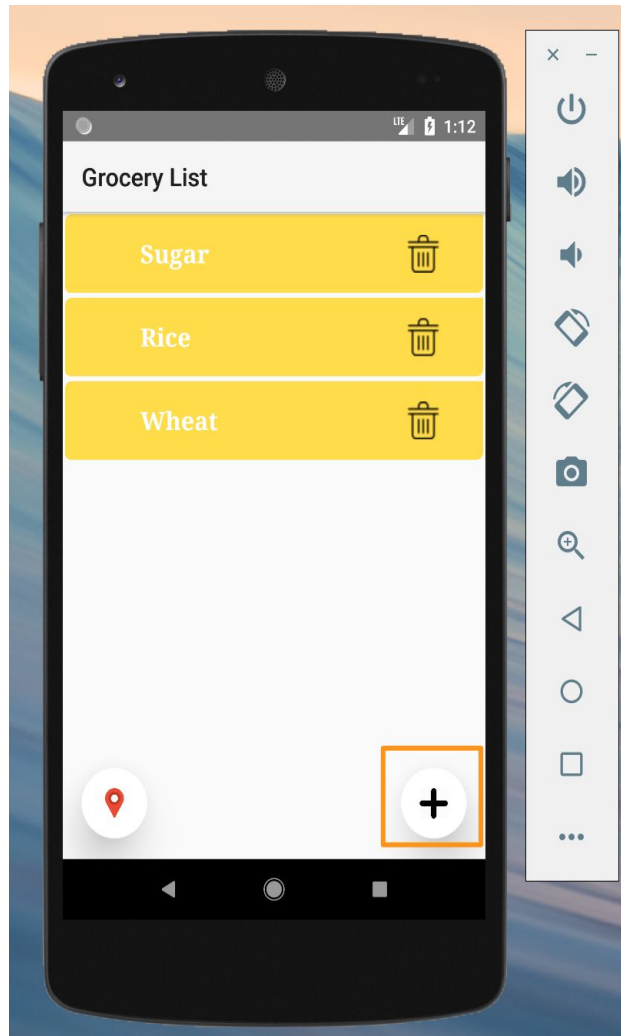
1. Select 'Grocery List' button on the Home screen



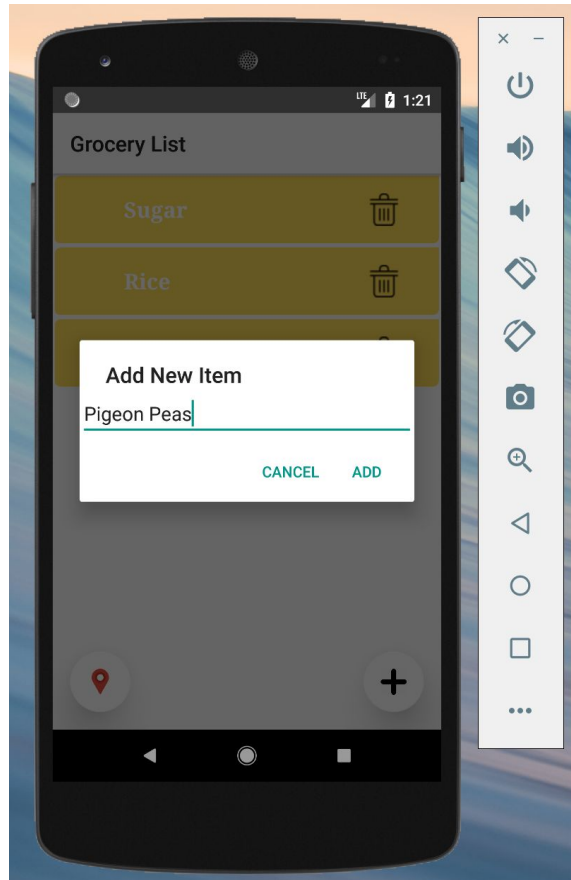
2. Displays the list of grocery items that is already added by the user.



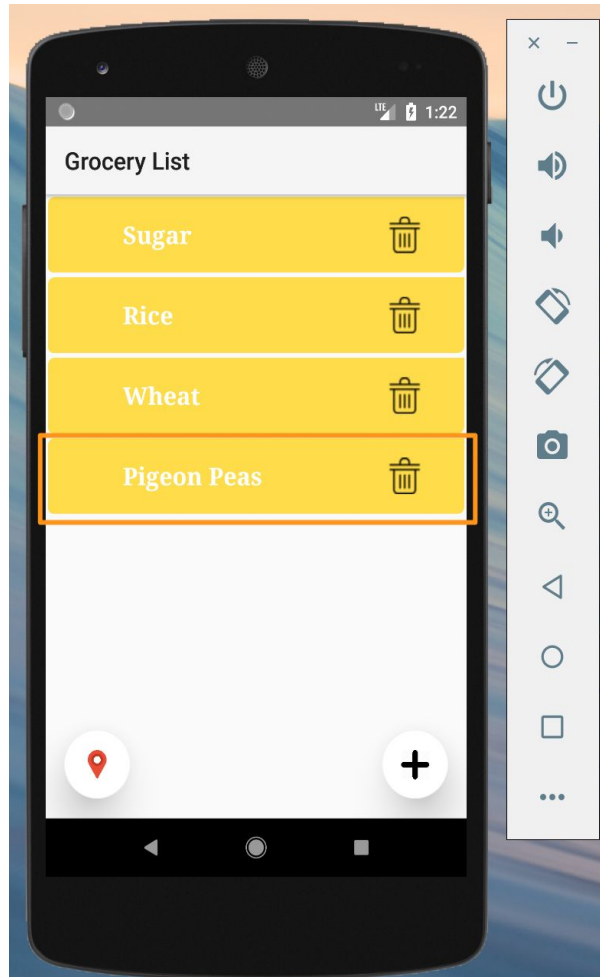
3. To add a new item to the grocery list, click on '+' button at the bottom.



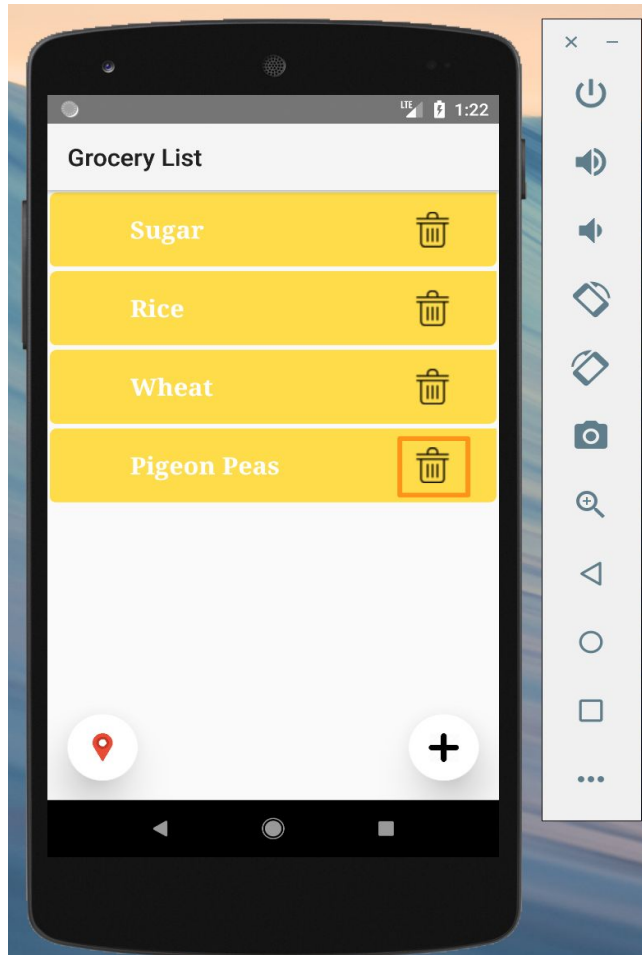
4. Add the new item and click 'ADD'.



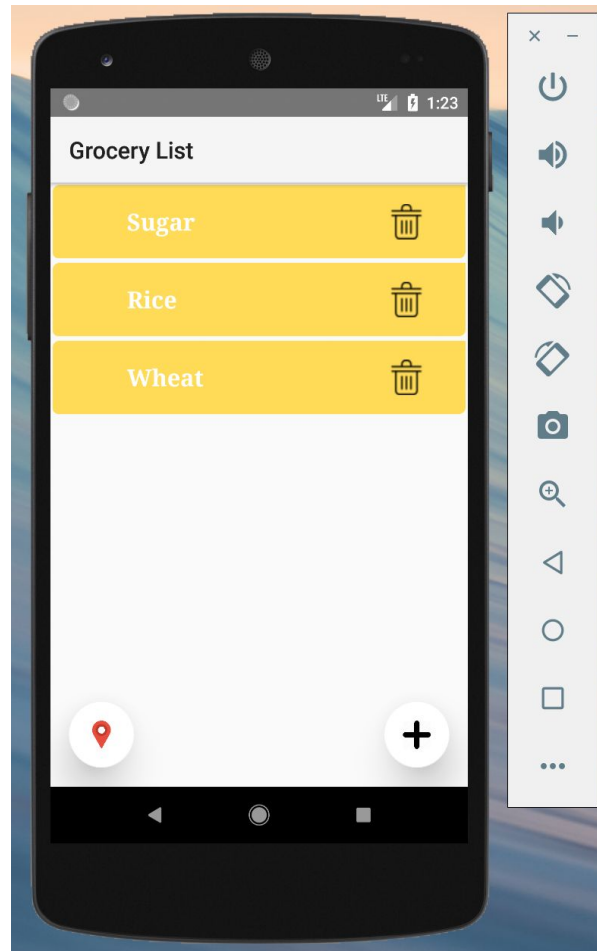
5. Check for the new item added into the grocery list.



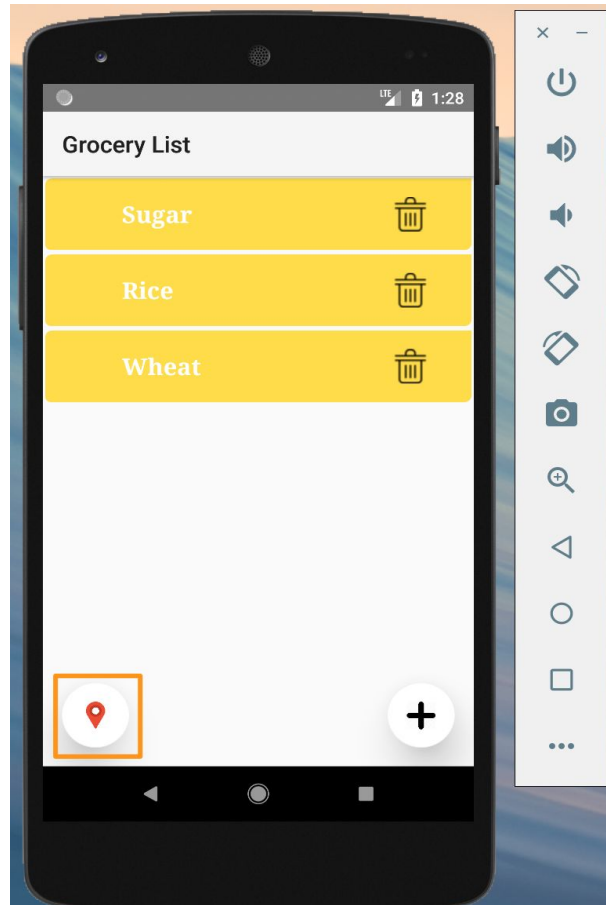
6. Delete the item by clicking on delete icon.



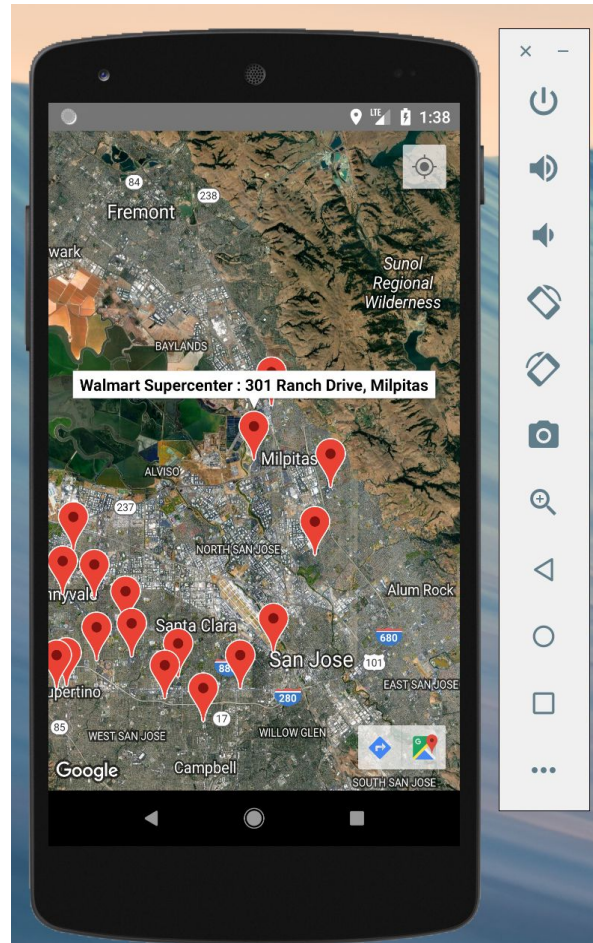
7. Item successfully deleted.



8. Click on 'Maps icon' to navigate to google maps to find nearby grocery stores.

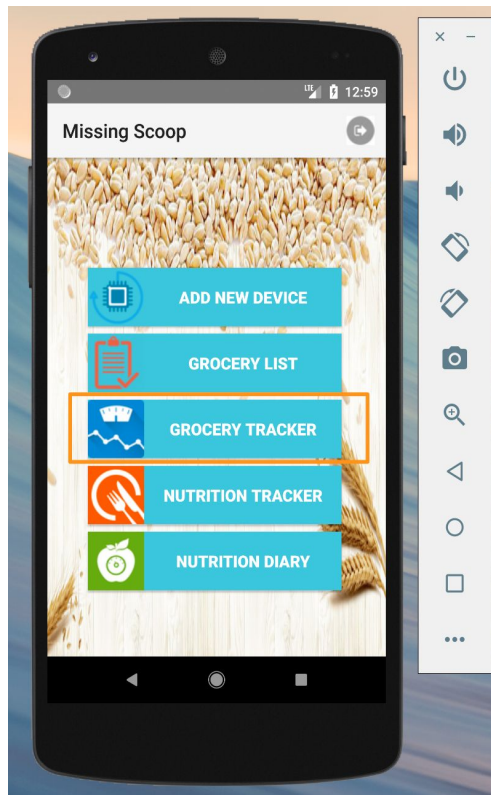


9. The next screen will take you to the maps showing the nearby groceries on google maps.

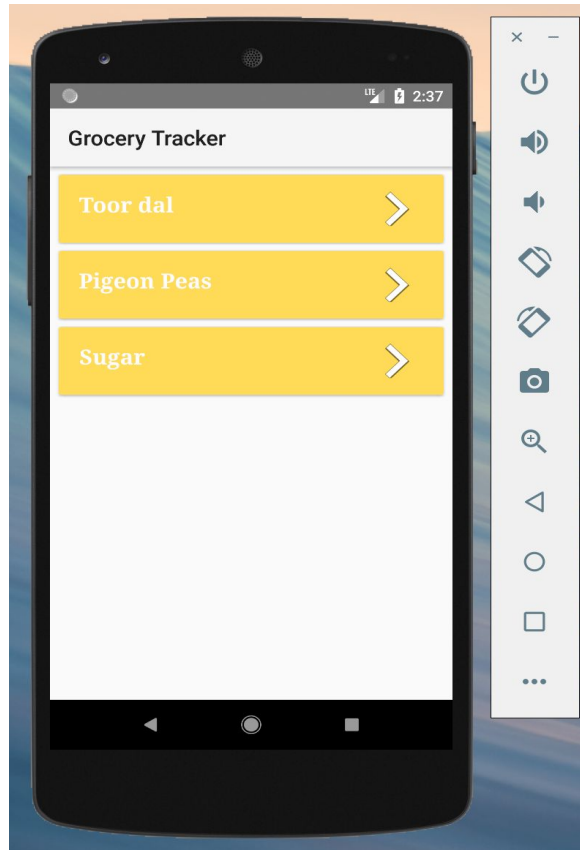


Grocery Tracker

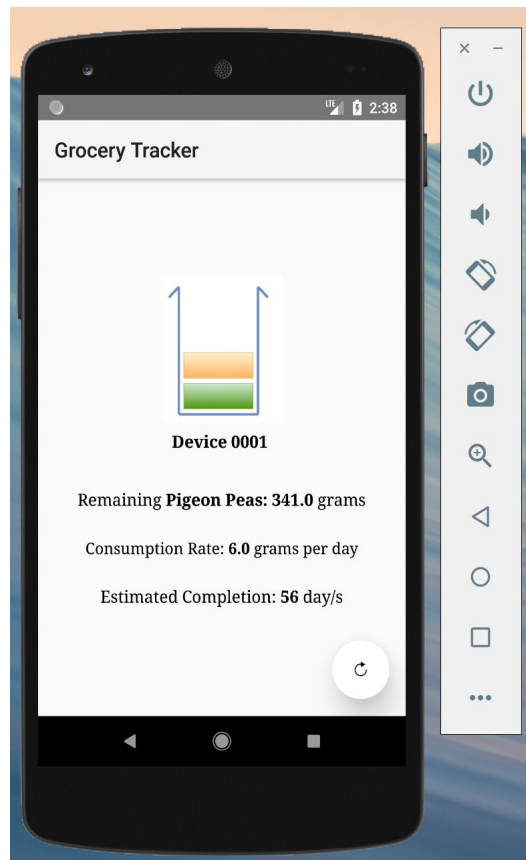
1. Select on 'Grocery Tracker' in Home screen.



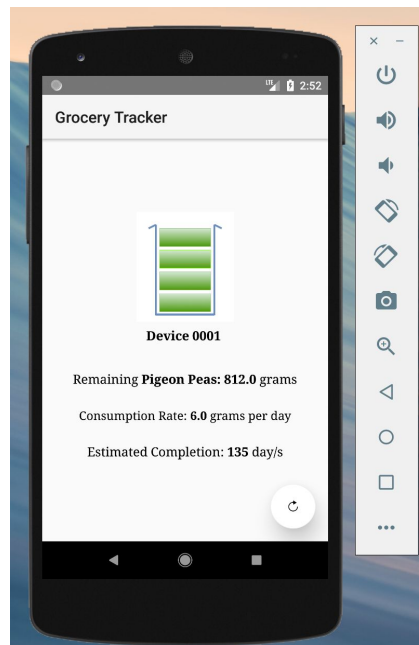
2. List the products that users have registered devices.



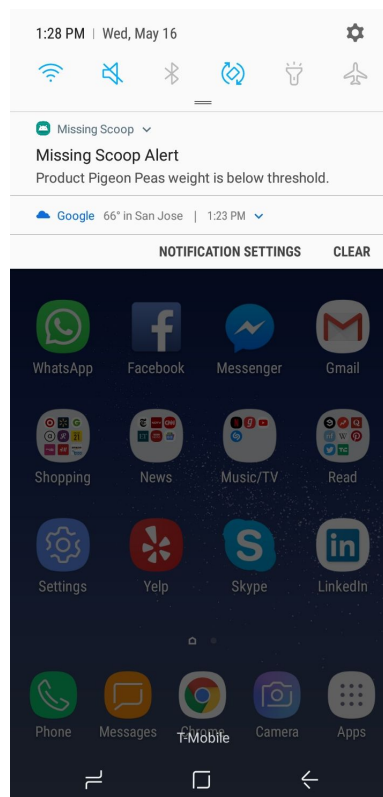
3. Select Pigeon Peas with which we have a device already registered with apt weight data. This will give you the current weight, consumption rate and estimated completion days.



3. When weight is full the screen looks as shown below.

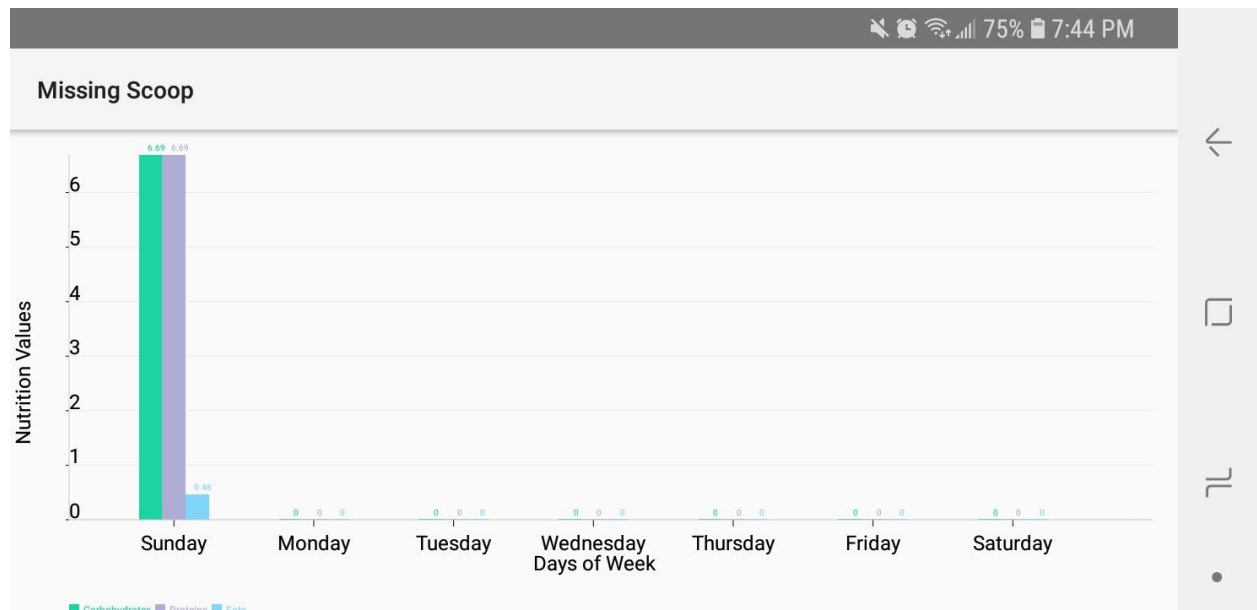


Push Notification



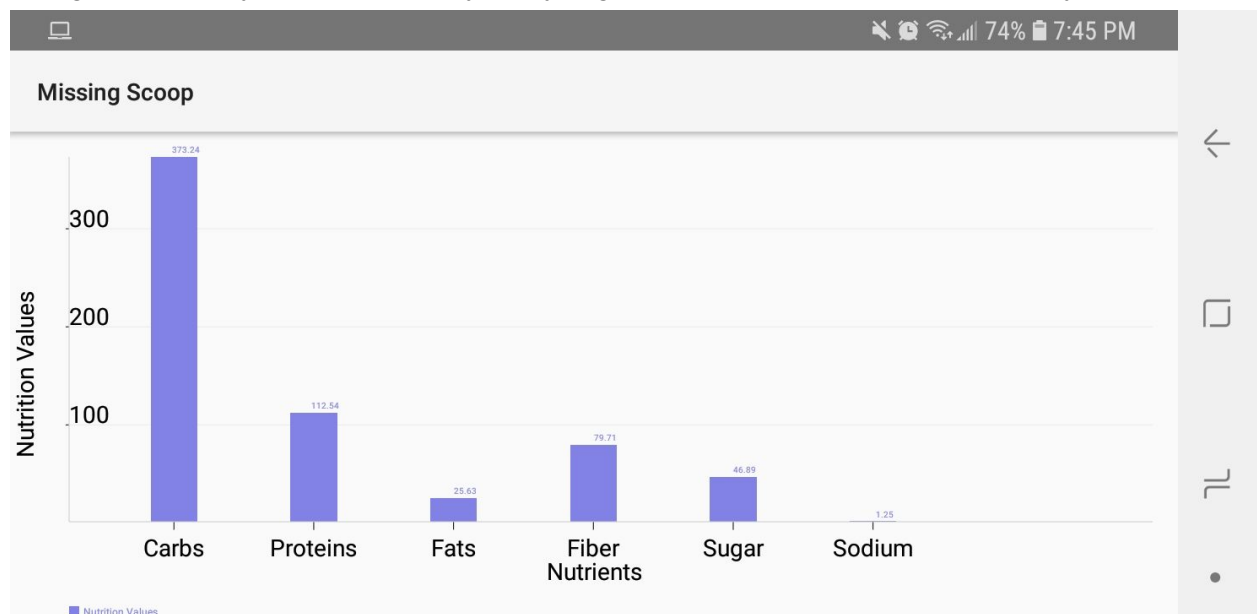
Nutrition Tracker

This gives the weekly nutrition chart by analysing the products consumed in that given week.



Nutrition Diary

This gives the daily nutrition chart by analysing the products consumed on that day.



4. System Testing and Experiment

4.1 Testing Scope

4.2 Automation Testing

Functional Testing

The functional test cases covers all possible positive scenarios.

4.2.1 Test Cases -GET Scenarios

Scenario 1:Verify device product mapping . Hit GET: <environment URL>/fetch/device/product?userName=<username>
Expected Result : Response code 200.

=====

Scenario 2:Verify daily nutrition . Hit GET: <environment URL>/nutrition/daily?userName=<username>
Expected Result : Response code 200.

=====

Scenario 3:Verify nutrition history. Hit GET: <environment URL>/nutrition/history?userName=<username>
Expected Result : Response code 200.

=====

Scenario 4:Verify all available nutrition. Hit GET: <environment URL>/fetch/nutrition/all
Expected Result : Response code 200.

=====

Scenario 5:Verify device weight. Hit GET: <environment URL>/fetch/device/weight?userName=<username>
Expected Result : Response code 200.

4.2.2 Test Cases -POST Scenarios

Scenario 1:Verify device product mapping is saved . Hit POST: <environment URL>/map/device/product
Payload:

```
{  
  "userName" : "<user name>",  
  "deviceId" : "< device id>",
```

```
"Label" : "<label>",  
"threshold" : "<threshold number>"  
}
```

Expected Result : Response code 200.

=====

Scenario 2:Verify add grocery to the list. Hit POST: <environment
URL>/add/grocery

Payload:

```
{  
"userName" : "<user name>",  
"grocery" : "<grocery name>"  
}
```

Expected Result : Response code 200.

4.2.3 Automation Setup

Include the following properties in POM.xml

```
<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
  <jetty.version>6.1.4</jetty.version>  
  <spring.version>4.2.4.RELEASE</spring.version>  
  <spring-boot.version>1.3.1.RELEASE</spring-boot.version>  
  <slf4j.version>1.7.21</slf4j.version>  
  <java.version>1.8</java.version>  
  <build.profile>default</build.profile>  
  <jackson.version>2.6.4</jackson.version>  
</properties>
```

Include the following dependencies in POM.xml

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>2.53.1</version>
</dependency>
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.1.1</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot</artifactId>
  <version>1.3.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>4.2.5.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.2.4.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>${jackson.version}</version>
</dependency>
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.9</version>
</dependency>
<dependency>
  <groupId>javax.ws.rs</groupId>
  <artifactId>javax.ws.rs-api</artifactId>
  <version>2.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpcore</artifactId>
  <version>4.4.5</version>
</dependency>

<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.2</version>
</dependency>
</dependencies>
```

4.2.4 Test Result

After running the test file as testng , retrieve the emailable report as following .

GET Scenarios:

Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
Default test	5	5	0	0	1.6 seconds		

Class	Method	# of Scenarios	Time (Msecs)
Default test — passed			
edu.sjsu.missingscoop_APITest.TestGETSenarios	getNutritionHistory ("GET_nutrition-history")	1	128
	getDeviceWeight ("GET_device-weight?deviceName=")	1	165
	getDailyNutrition ("GET_nutrition-daily")	1	127
	getAllNutrition ("GET_nutrition-all")	1	129
	getDeviceProductMap ("GET_device-product?deviceName=")	1	987

Default test

edu.sjsu.missingscoop_APITest.TestGETSenarios:getNutritionHistory

[back to summary](#)

edu.sjsu.missingscoop_APITest.TestGETSenarios:getDeviceWeight

[back to summary](#)

edu.sjsu.missingscoop_APITest.TestGETSenarios:getDailyNutrition

[back to summary](#)

edu.sjsu.missingscoop_APITest.TestGETSenarios:getAllNutrition

[back to summary](#)

edu.sjsu.missingscoop_APITest.TestGETSenarios:getDeviceProductMap

[back to summary](#)

POST Scenarios:

Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
Default test	2	2	0	0	1.4 seconds		

Class	Method	# of Scenarios	Time (Msecs)
Default test — passed			
edu.sjsu.missingscoop_APITest.TestPOSTScenarios	addGrocery ("POST_add_grocery")	1	340
	saveDeviceProductMap ("POST_map_device_product")	1	1016

Default test

edu.sjsu.missingscoop_APITest.TestPOSTScenarios:addGrocery

[back to summary](#)

edu.sjsu.missingscoop_APITest.TestPOSTScenarios:saveDeviceProductMap

[back to summary](#)

4.3 Manual Testing Scenarios

The team validated the app overall functionality by performing a round of feature testing after developing test cases . Test cases are outlined in table below.

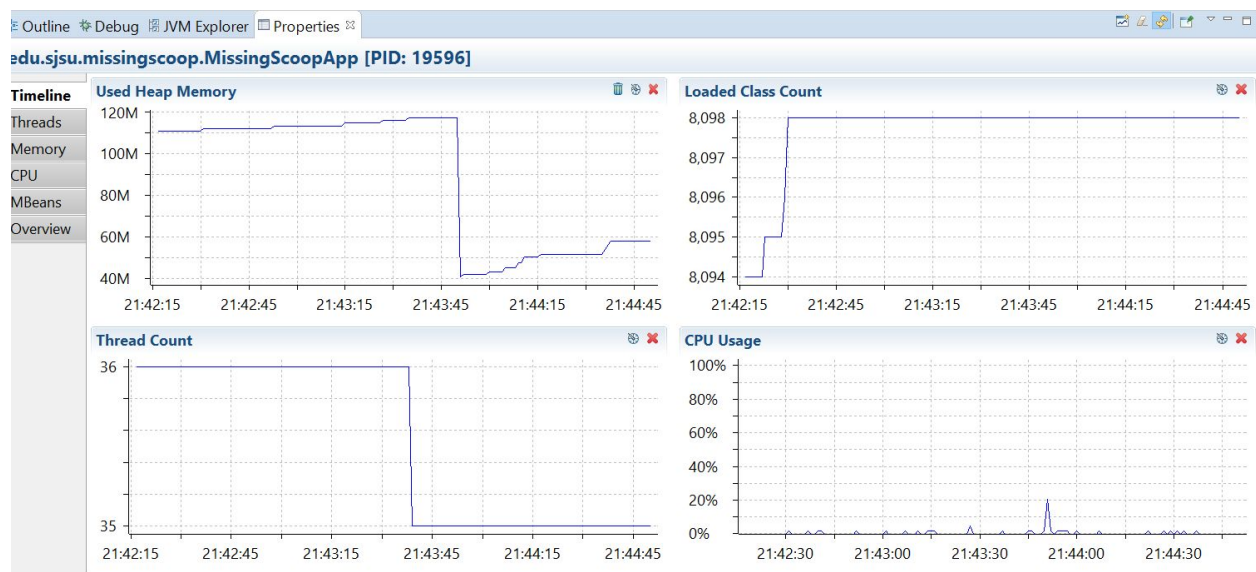
ID	Test Case	Expected Result	Actual Result	Defects
TC-01	Verify user can sign up	User signup successfully	Pass	None
TC-02	Verify user can login to application	User login successfully	Pass	None
TC-03	Verify user can redirect to the corresponding page after picking from menu	User redirected successfully	Pass	None
TC-04	Verify existing user can add grocery to the	New grocery shows in grocery list successfully	Pass	None

	grocery list			
TC-05	Verify user choose to add to grocery list but wish to cancel	User has an option and can successfully cancel	Pass	None
TC-06	Verify new user can add to grocery list	New user should see an empty list and can successfully add grocery	Pass	Found one defect and fixed right away
TC-07	Verify user can delete each grocery from the list	User successfully delete from grocery list	Pass	Found one defect and fixed right away
TC-07	Verify the user can find nearby grocery store	User successfully launch google map and it shows nearby stores	Pass	None
TC-08	Verify user can track grocery consumption with grocery tracker	User successfully see remaining , consumption rate and estimated completion data after picking the grocery	Pass	None
TC-09	Verify user can view weekly nutrition chart by picking nutrition tracker	The chart will be successfully displayed to user	Pass	None
TC-10	Verify user can view daily nutrition by picking nutrition diary	The chart will be successfully displayed to user	Pass	None
TC-11	Verify user can	User	Pass	None

	add new device by using QR Code Scanner	successfully add device		
TC-12	Verify user can logout	User logout successfully	Pass	None

4.4.Profiling

We used JVM Monitor 3.8.1 as a profiling tool which helps measuring performance factors such as CPU usage of our application . This will further helps us to find memory leakage in our app .This can be extended by using Jmeter to hit our app multiple times to simulate users and then monitor usage to find out the benchmark for handling number of TPS for our app. This will help us to find out areas of improvements to boost up the performance of our app.



5. Conclusion and Future Work

5.1 Project Summary

Grocery products like rice, wheat, cereals etc., are must have and are everyday part of the kitchen. Especially, it is very important for growing babies at home to have a healthy diet. Manual tracking of the groceries and baby food is a pain. The Android app 'Missing Scoop' developed as part of the project provides a platform for the user to keep a track of groceries and baby food. In addition, it acts as a single stop mobile application with following features:

- **IoT interface to track the grocery products:** The app user can add a new IoT weight sensor set up and register this new device with our app. This can be done by scanning the qr code of the device using camera. After which the user can keep a track of the current weight of the product, consumption rate and estimated completion days.
- **Push notifications:** When user tracked grocery goes below threshold, user will be notified.
- **Manage grocery list:** Missing Scoop provides a platform for the users to manage the list of groceries. The user can add, delete and display the list.
- **Check for nearby grocery stores on maps:** The user can also access google maps from the app to check out for nearby grocery stores. User can also use the google maps features of navigation from our app.
- **Weekly nutrition tracker:** The user can evaluate the nutrition he/she has consumed on that specific day. The nutrition values obtained are Carbohydrates, Proteins, Fats, Fiber, Sugar and Sodium.
- **Daily nutrition tracker:** The user can evaluate the nutrition he/she has consumed on the current week. The nutrition values obtained are Carbohydrates, Proteins and Fats.

5.2 Individual Contributions

Tasks	Contribution
Hardware Setup	Ashwini Shankar Narayan
AWS IoT configuration	Ashwini Shankar Narayan
Grocery Tracking – Backend REST API	Ashwini Shankar Narayan
Grocery Tracking – Frontend Android Activity	Ashwini Shankar Narayan
Google Maps to find Nearby groceries	Ashwini Shankar Narayan
Home Screen Activity on Android	Ashwini Shankar Narayan
Device Product Mapping - Backend REST API	Anushri Srinath Aithal
Device Product Mapping Android UI	Anushri Srinath Aithal
QR Code Scanning with Google Visions	Anushri Srinath Aithal

Volley REST Client Setup for Android	Anushri Srinath Aithal
Nutrition Tracking Backed API and Cron Job	Anushri Srinath Aithal
Nutrition Tracking Android UI with AChartEngine	Anushri Srinath Aithal
Signup, Login and Logout with Firebase	Anushri Srinath Aithal
Push Notification Backend with Google Cloud Messaging	Anushri Srinath Aithal
Push Notification on Android	Anushri Srinath Aithal
Grocery Shopping List Backend API	Mojdeh Keykhanzadeh
Grocery Shopping List Android UI	Mojdeh Keykhanzadeh
UI Mockups for all Android Screens	Mojdeh Keykhanzadeh
Automation Testing Framework Setup and actual testing	Mojdeh Keykhanzadeh
Manual test cases	Mojdeh Keykhanzadeh
Profiling of the project	Mojdeh Keykhanzadeh

5.6 Project Deployment Instructions

REST Backend Project Deployment on Local

1. Clone this project
https://github.com/shriaithal/missingscoop/tree/master/MissingScoop_RestEndpoint
2. Run command "mvn clean install" on the folder where pom.xml is present
3. To execute the project from eclipse Right click on the project -> Run as Java Application
4. To execute on command line run the command java -jar "jar name"

REST Backend Project Deployment on EC2

1. Run "mvn clean install" command on the project parent folder
2. Copy the jar from user .m2 folder to ubuntu EC2

3. Setup a service to run spring boot application using the below commands

```
Chown x+ missingscoop.jar  
sudo ln -s /path/to/missingscoop.jar /etc/init.d/missingscoop  
update-rc.d missingscoop defaults 95
```

4. Start service using the below command

```
sudo service missingscoop start
```

Android Project Deployment

1. Clone project from
https://github.com/shriaithal/missingscoop/tree/master/MissingScoop_AndroidApp
2. Make Project on Android Studio
3. Use the APK to run on Simulator or on Android Phone

Hardware Setup

Instructions provided under Hardware Architecture of this report

Test Automation Setup

1. Clone project
https://github.com/shriaithal/missingscoop/tree/master/MissingScoop_APITest
2. Run “ mvn clean install -DskipTests” on the folder where pom.xml is present
3. Add TestNg to Eclipse
4. Import project to Eclipse
5. Make sure to replace environment URL with the URL and user name with the username
6. Run as TestNg
7. To run from command line, run “mvn test”

5.4 Future Work

The Missing Scoop app can be further extended to provide more functionality such as integrating with fitness tracking apps, providing healthy choices to users based on consumption analysis and integrating with weight watchers app to allow users to adjust their diet .

The other area that the app can be extended to is supply chain management. Retail industry will be our target audience to extend the app . The app will provide real time data to retailers to allow them efficiently fill up their inventory. The data can be further analyzed to give detailed information of popular items that get sold faster. This will boost up and optimize sales.

