IoT Enabled Smart Inventory Design Review

`

Adil Saldanha | Ammar Rehan | Melika Salehi | Wency Go

24 November 2019

Group 110

Contents

Requirements Document

The requirements, constraints, and goals for this project are summarized in the table below. The RCGs were obtained and clarified through meetings with the client and was presented during the project proposal. They are available here as a reference for the design rationale throughout this document.

REQUIREMENTS		
COMPONENT	**REQUIREMENT**	**DESCRIPTION**
Weight Sensor	Sensor must be able to handle an upper limit of 700g -2250g (up to 150 items).	The client has said they want the container to be able to hold up to 150 items weighing between 5g and 15g. With this information the container should be able to measure up to 2.25kg
End Device(s)	Must communicate to edge device(s) via a low bandwidth communication protocol	The client has asked for an end device that can wirelessly communicate with the edge device to update weight sensor, hence inventory data. Live updating of data is not required.
	Configuration of product weight and type.	The client requires end devices to be usable for different products hence it needs parameters like weight to be adjustable.
Edge Device (s)	Must communicate to end device(s) via a low bandwidth protocol.	Receiving data from end devices requires compatible protocol
	Must communicate to cloud via a high bandwidth protocol.	Syncing with Azure IoT Hub requires active WiFi connection

Table 1: Requirements for the project and its description.

CONSTRAINTS		
COMPONENT	**CONSTRAINT**	**DESCRIPTION**
Weight Sensor	Sensor must have 1 mg of sensitivity.	This is to mitigate any variation in product mass. For example, 3 packets of a product stated at 5 grams might weigh 5.1g, 5.03g and 4.98g. This level of sensitivity will mitigate that error.
Data Transfer	Data transfer between end and edge devices must be wireless.	This is so end devices can serve untethered since store layout may vary and since wires add clutter and are a tripping hazard.
	Data transfer between edge devices to cloud must be wireless.	Allows end user to place edge device anywhere within their store.
Cost	Production cost of final product must be kept as low as possible	This does not include prototyping costs where the client is willing to provide some resources.

Table 2: Constraints for the project and its description.

GOALS		
COMPONENT	**GOAL**	**DESCRIPTION**
End product	Scalable platform for production and future development	E.g. addition of new sensors in the future E.g. multiple device integration
	Simplistic	Usable with minimal to no training
	Robust data handling	Able to handle interruptions in data processing

Table 3: Goals for the project and its description.

Design Document

Overview

The IoT-enabled Smart Inventory tracks inventory by collecting data in the form of product weight, transmitting this data from end devices to an edge device, which finally sends this data to the cloud, where the user may then access their inventory data. The requirements, constraints, and goals (RCGs) for the project are available in Appendix A and can be referenced with respect to our design rationale.

Two key terms used in this document are the 'end device' and 'edge device'. In Internet of Things (IoT) infrastructure, an **end device** is primarily used to collect data and transmit this data through some form of low bandwidth communication protocol for low power consumption and size constraints. The **edge device** essentially acts as a middleman, or router, to receive data from the end device and then send this data to the cloud, where developers and users may access the data.

The three major components of our system are as follows:

1. End Device - Arduino Uno connected with load sensor(s) and a Zigbee Transceiver.
2. Edge Device - Raspberry Pi 3+ with a Zigbee Transceiver and WiFi adapter.
3. Cloud Computing - Microsoft Azure IoT Hub

Each of these components will be explained in more detail in their own section of this document as well as the design rationale for choosing these components.

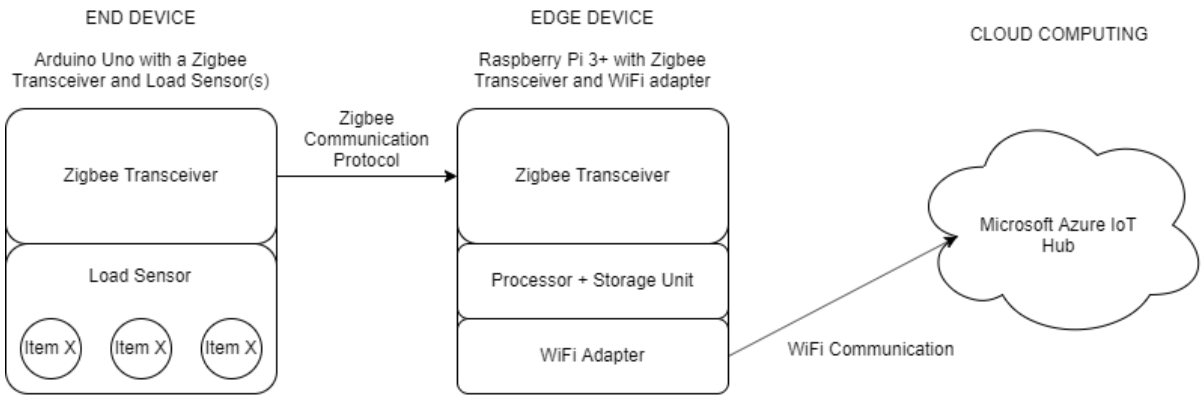

Figure 1 System Architecture diagram outlining the major components and communication protocol between these components.

End Device

The main processing unit for our end device was chosen to be an Arduino Uno, connected with a micro load cell and an Xbee transceiver. The Arduino family of electronics conveniently provides a microprocessor, on-board memory, I/O pins, and an IDE, making it an easy choice for the design. Another important reason is that there is a Zigbee communication library available. Without software libraries, we would have to develop our own library for processing and transmitting data between end and edge devices which would significantly increase the time required for this project. In addition, most of our team is familiar with coding in Arduino so using the Arduino IDE will allow us to write and debug the code more easily than writing in other low-level programming languages, as well as code readability for future developers and stakeholders. We have also chosen the Arduino Uno for scalability purposes. The board also contains numerous pins where we can conveniently attach components such as the load sensor and the Xbee transceiver. If, in the future, it is required to add more components, it is easily accomplished.

Low Bandwidth Communication

To clarify, Zigbee is the name of the low bandwidth communication protocol we are using to transmit data and Xbee is a popular brand of electronics components that utilize the Zigbee protocol. Our Xbee transceiver was chosen because it was compatible with the Arduino Uno and the Raspberry Pi used as the edge device.

Zigbee was chosen over the alternatives due to its ideal range and its ability to create a Zigbee ‘mesh network’ so that a multitude of Zigbee-enabled devices can communicate with one another. Compared to Bluetooth, which has a much smaller range at around 10 meters, Zigbee can range up to 100 meters and would be able to operate in small- to medium-sized warehouses or stores and track multiple end devices at a time. Another alternative was LoRa technology, but its range was far too large, up to 2-3-kilometer coverage, which has the potential to create interference with other IoT-enabled Smart Inventory systems in that area.

Load Cell

For our choice of weight sensor, we decided to go with a 780g micro load cell. The reasoning behind this was that it was the most accurate device for what it costs and has an appropriate range for our purpose. Other options such as strain gauges were more expensive or measured much higher masses which would induce more error. The load cell only measures up to 780g, whereas our requirements state that we want to measure up to 2.25kg. To achieve this installing a combination of 3 load cells in one container would work although this would drive up cost considerably, so these would be something to consider for milestone 3. One concern with this load cell is that the connection wires are not very strong so this will also have to be considered when doing the final design.

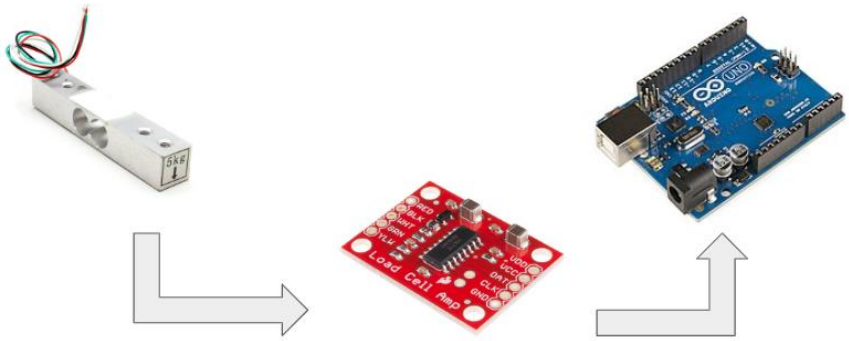

Figure 2 Load cell connection to HX711 board to Arduino Uno

For our setup, we decided to go with connecting the load cell to a HX711 board which is then connected to the Arduino. The HX711 board is an amplifier which reads the changes in resistance, amplifies them and sends the readings to the Arduino. One of the reasons we picked the HX711 is that there is a lot of existing support for the board such as Arduino libraries for reading and calibrating the load cell. Another reason is that the board is relatively inexpensive.

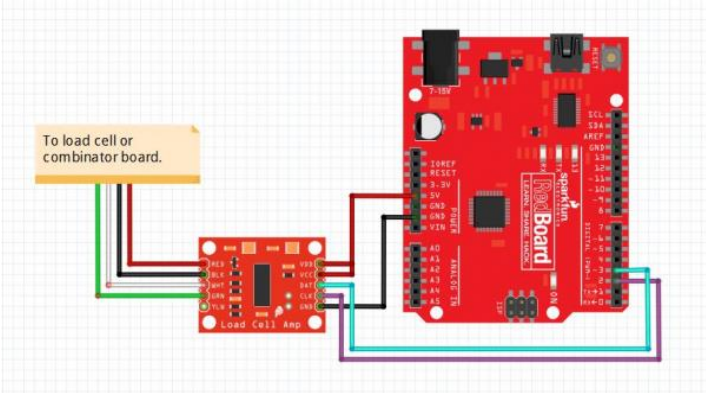

Figure 3 Load cell connection setup for connecting load cell to HX711 board to a microprocessor

The microprocessor used in this diagram is not an Arduino but for our design, we are using an Arduino Uno [1]

[1] Image *source: https://learn.sparkfun.com/tutorials/load-cell-amplifier-hx711-breakout-hookup-guide/al*

Edge Device

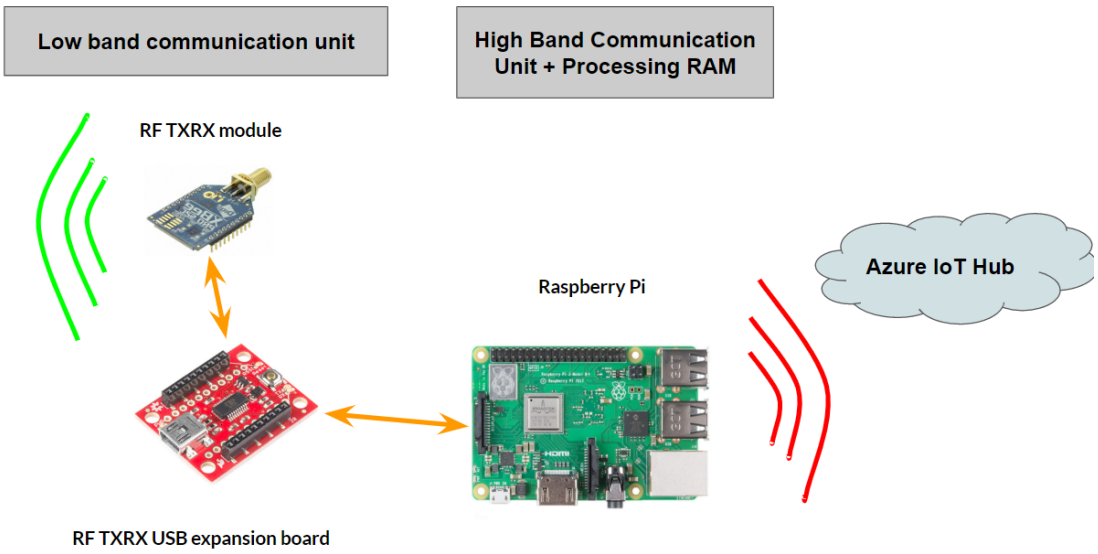

Figure 4 Edge device connectivity and communications

The edge device architecture consists of 3 major modules: the Raspberry Pi 3 B+, XBee-Pro ZigBee communication module, and XBee serial interface. The parts, their purpose, and the design rationale are summarized in Table 1.

Table 1 Edge device architecture component details

COMPONENT	PURPOSE	DESIGN RATIONALE
Raspberry Pi 3 B+	Main processing unit. Acts as a host gateway for connecting to the Azure cloud	WiFi compatibility Available on-board ports Available Azure libraries Economical price
Xbee-Pro ZigBee communication module	RX/TX for Zigbee protocol, used for communicating with end devices.	The same model is being used on end device(s). Having the same model at both ends allows for a simpler, hassle free implementation.
Xbee Explorer USB	Serial interface between the XBee-Pro ZigBee module and the Raspberry Pi 3 B+	Available hardware interface for Raspberry Pi.

System & Software Architecture

The edge device will act as the host gateway between the end device and the cloud. The Rapsberry Pi, equipped with both the Xbee ZigBee communication module and a WiFi adapter, allows for receiving the end device data, pre-processing the data with its on-board processor, and finally submitting the pre-processed data to the cloud for further use. The top-level software architecture of this process flow can be seen in Table 2.

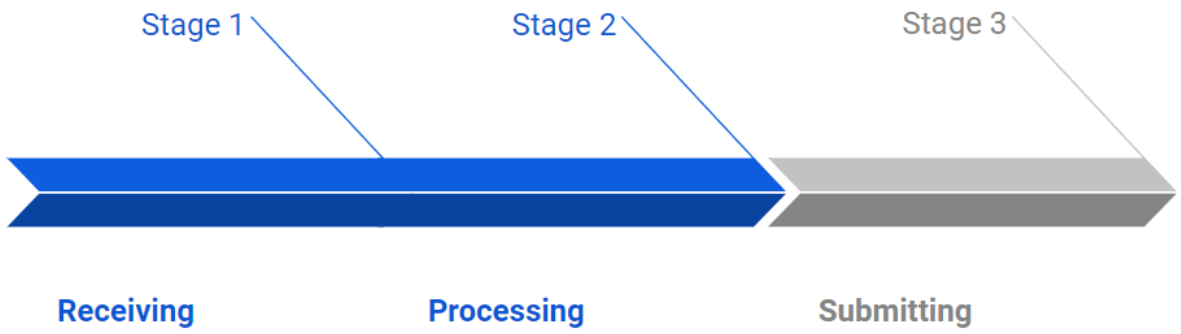

Figure 5 Edge device process flow

Table 2 Edge device data managements architecture details

STAGE	PURPOSE	RATIONALE
ISR	The interrupt service routine prioritizes accepting incoming data.	Data submission to cloud can be done at any time, as the raspberry pi can hold the data in memory.
VALIDATION	To check for legitimacy of incoming data. The end device will add an identification key to the data being forwarded.	The nature of data is not sensitive, and the received data is not encrypted. This is to avoid picking up data from nearby devices that are not end devices.
PREPARATION	Cleans up the data string associated with each read, so it can be sent to the cloud.	The data received will contain extra information due to the protocol being utilized. This extracts the information we care about.
SUBMISSION	Starts an API request to the cloud. Proceeds to submit data.	The Raspberry Pi is not in constant active connection with the Microsoft Azure IoT Hub. This is due to the limited number and duration of API requests that are given to cloud users, depending on the contract.

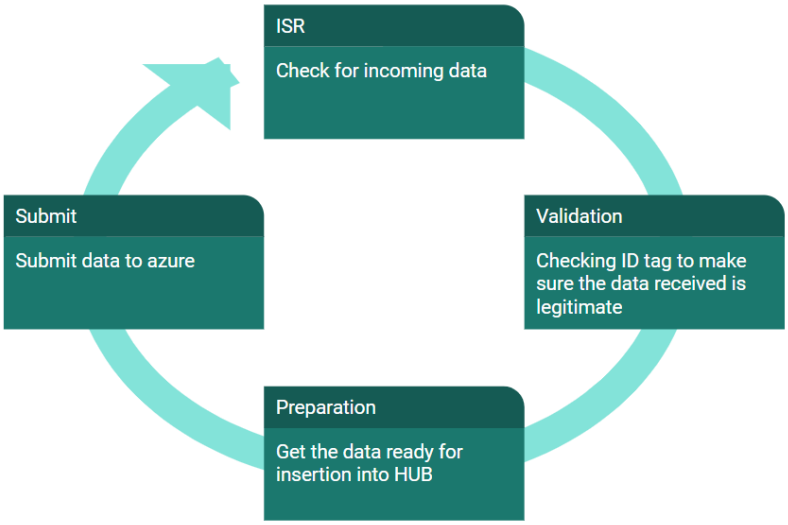

Figure 6 Data management cycle in the edge device

Once the edge device sends the data to the cloud, we now use Microsoft Azure IoT Hub to access the data for further use. The Microsoft Azure IoT Hub is an online, cloud database that contains each edge device connected to it. Each edge device is registered with a unique identifier. Once it is registered, the IoT Hub and the device can then transfer data between each other. The Microsoft Azure IoT Hub was chosen because of client requirements – their current system uses Microsoft Azure for their cloud computing, so it was chosen for scalability and integration purposes.

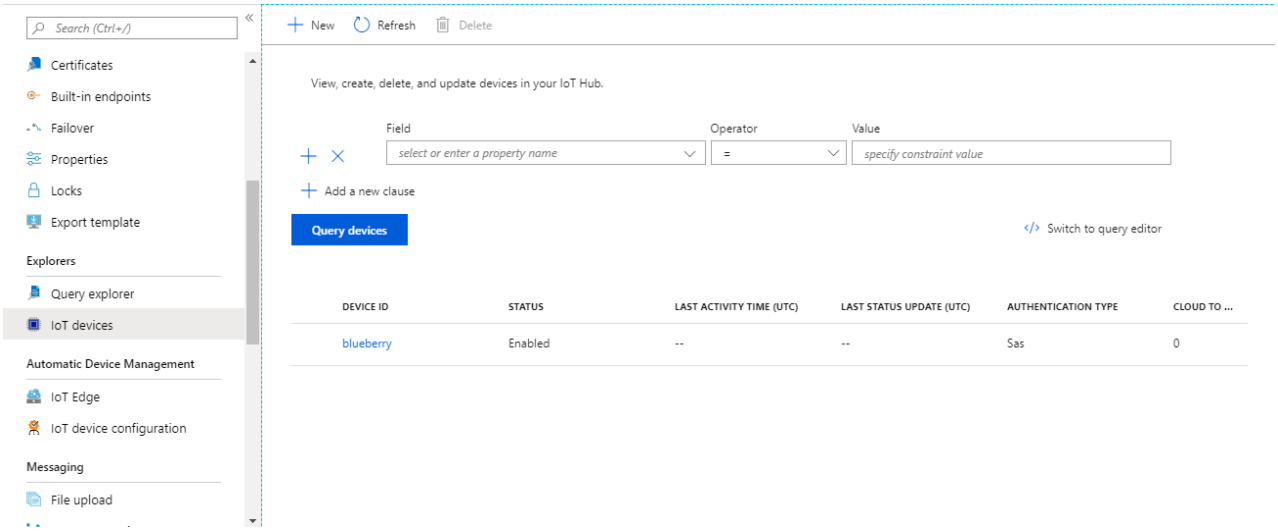

Figure 7 Microsoft Azure IoT Hub web interface with a sample device registered as 'blueberry'

Validation Document

Testing and validation is essential for any software-based projects, especially when collecting analog data from hardware and sending this data through multiple abstraction layers. Outlined and summarized below is our validation methods for each component of the project.

Table 3 System test and validation breakdown by major component

Component	Requirement	Testing	Validation
End Device	User configuration of product parameters	Changing device parameters and comparing observed versus expected results	Data can be accessed via edge device and cloud
Low Bandwidth Communication	Wireless communication	Use Xbee modules to identify transmitted and received data	Check that the correct data is being sent and received by the corresponding end and edge devices
	Scalability for multiple devices	Adding multiple end devices to communicate with a single edge device	Check that the data is tagged by each end device and the correct data is associated with the correct end device
Load Cell	Must be able to measure up to 150 items ranging from 7-15g each (total up to 2.25 kg) with ± 1mg accuracy	Weigh items using the load cell and compare this measurement with other weighing scales.	Check that the data matches with commercially available weighing scales.
Edge Device	Send data to cloud	Send data via a sample script provided by Microsoft.	View the data using Microsoft Azure IoT Hub, which can display data transfer in real time.
	Receive data via ZigBee protocol	Run a script to check timely reception via ISR process	It can be visually confirmed for short strings. Can also use MS excel compare cell value functions
	Must be able to operate for 8-10 hours without interruption	Test continues run as well as wifi downtime handling	Confirmation can be made through Microsoft Azure IoT Hub web interface.

Appendix

Appendix A: Sample Code for Arduino ZigBee Communications

A sample base code written by our team for receiving data with an Xbee-connected Arduino with an Xbee module connected to a computer and sending data via XCTU software.

```
int analogPin = A0;
int val = 0;

void setup() {
  // initialize serial communication
  Serial.begin(9600);
}

void loop() {
  // read analog value at given pin
  val = analogRead(analogPin);
  //send read value at pin (calibration required)
  Serial.println(val);
  delay(1000);
}
```

Appendix B: Microsoft Sample Code for IoT Hub and Raspberry Pi

Sample code provided by Microsoft to test Rapsberry Pi communication with the Microsoft Azure IoT Hub with a small change to the device name to ‘blueberry’.

```
/*
* IoT Hub Raspberry Pi C - Microsoft Sample Code - Copyright (c) 2017 -
Licensed MIT
*/
#include "./wiring.h"

static unsigned int BMEInitMark = 0;

#if SIMULATED_DATA
float random(int min, int max)
{
    int range = (int)(rand()) % (100 * (max - min));
    return min + (float)range / 100;
}

int readMessage(int messageId, char *payload)
{
    float temperature = random(20, 30);
    snprintf(payload,
             BUFFER_SIZE,
             "{ \"deviceId\": \"blueberry\", \"messageId\": %d,
\"temperature\": %f, \"humidity\": %f }",
             messageId,
             temperature,
             random(60, 80));
    return (temperature > TEMPERATURE_ALERT) ? 1 : 0;
}

#else
int mask_check(int check, int mask)
{
    return (check & mask) == mask;
}

// check whether the BMEInitMark's corresponding mark bit is set, if not, try
to invoke corresponding init()
int check_bme_init()
{
    // wiringPiSetup == 0 is successful
    if (mask_check(BMEInitMark, WIRINGPI_SETUP) != 1 && wiringPiSetup() != 0)
    {
        return -1;
    }
    BMEInitMark |= WIRINGPI_SETUP;

    // wiringPiSetup < 0 means error
    if (mask_check(BMEInitMark, SPI_SETUP) != 1 &&
wiringPiSPISetup(SPI_CHANNEL, SPI_CLOCK) < 0)
```

```
    {
        return -1;
    }
    BMEInitMark |= SPI_SETUP;

    // bme280_init == 1 is successful
    if (mask_check(BMEInitMark, BME_INIT) != 1 && bme280_init(SPI_CHANNEL) !=
1)
    {
        return -1;
    }
    BMEInitMark |= BME_INIT;
    return 1;
}

// check the BMEInitMark value is equal to the (WIRINGPI_SETUP | SPI_SETUP |
BME_INIT)

int readMessage(int messageId, char *payload)
{
    if (check_bme_init() != 1)
    {
        // setup failed
        return -1;
    }

    float temperature, humidity, pressure;
    if (bme280_read_sensors(&temperature, &pressure, &humidity) != 1)
    {
        return -1;
    }

    snprintf(payload,
             BUFFER_SIZE,
             "{ \"deviceId\": \"blueberry\", \"messageId\": %d,
\"temperature\": %f, \"humidity\": %f }",
             messageId,
             temperature,
             humidity);
    return temperature > TEMPERATURE_ALERT ? 1 : 0;
}
#endif

void blinkLED()
{
    digitalWrite(LED_PIN, HIGH);
    delay(100);
    digitalWrite(LED_PIN, LOW);
}

void setupWiring()
{
    if (wiringPiSetup() == 0)
    {
        BMEInitMark |= WIRINGPI_SETUP;
    }
    pinMode(LED_PIN, OUTPUT);)
```

Appendix C: Load cell test and calibration sample code

Load cell Source Code

Basic Load cell test:
Source: https://github.com/sparkfun/HX711-Load-Cell-Amplifier/tree/master/firmware

```
#include "HX711.h"

// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = A0;
const int LOADCELL_SCK_PIN = A1;

HX711 scale;

void setup() {
  Serial.begin(57600);
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
}

void loop() {

  if (scale.is_ready()) {
    long reading = scale.read();
    Serial.print("HX711 reading: ");
    Serial.println(reading);
  } else {
    Serial.println("HX711 not found.");
  }

  delay(1000);

}
```

Load cell calibration

Source: https://github.com/sparkfun/HX711-Load-Cell-Amplifier/tree/master/firmware

```
#include "HX711.h"
#define LOADCELL_DOUT_PIN  3
#define LOADCELL_SCK_PIN  2


HX711 scale;
float calibration_factor = -7050; //-7050 worked for my 440lb max scale setup


void setup() {
  Serial.begin(9600);
  Serial.println("HX711 calibration sketch");
  Serial.println("Remove all weight from scale");
  Serial.println("After readings begin, place known weight on scale");
  Serial.println("Press + or a to increase calibration factor");
  Serial.println("Press - or z to decrease calibration factor");
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
  scale.set_scale();
  scale.tare();   //Reset the scale to 0


  long zero_factor = scale.read_average(); //Get a baseline reading
  Serial.print("Zero factor: "); //This can be used to remove the need to
tare the scale. Useful in permanent scale projects.
  Serial.println(zero_factor);
}

void loop() {

  scale.set_scale(calibration_factor); //Adjust to this calibration factor

  Serial.print("Reading: ");
  Serial.print(scale.get_units(), 1);
  Serial.print(" lbs"); //Change this to kg and re-adjust the calibration
factor if you follow SI units like a sane person
  Serial.print(" calibration_factor: ");
  Serial.print(calibration_factor);
  Serial.println();

  if(Serial.available())
  {
    char temp = Serial.read();
    if(temp == '+' || temp == 'a')
      calibration_factor += 10;
    else if(temp == '-' || temp == 'z')
      calibration_factor -= 10;
  }
}
```

Appendix D: Microsoft Azure Web Interface

Microsoft Azure is a cloud computing service provided by Microsoft. One of the services we are particularly interested in is the IoT Hub, which is a cloud database that communicates with and tracks IoT devices. It has a developed web interface that allows the user to generate identifier keys for registering new IoT devices and metrics to display usage statistics and data being transferred.

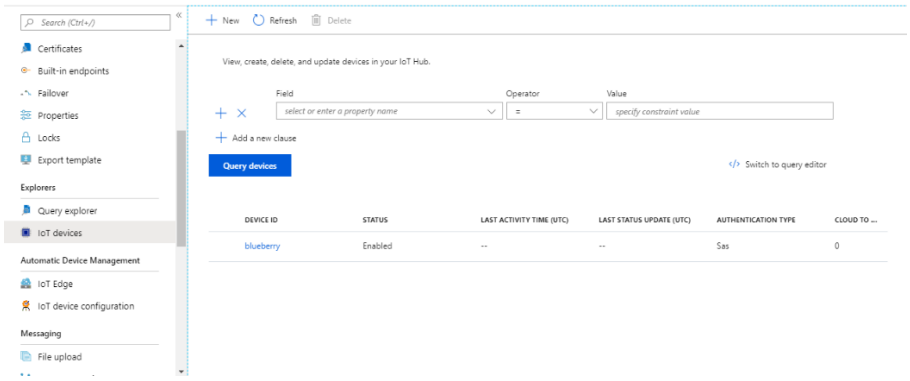

Figure 1: Microsoft Azure IoT Hub device list.

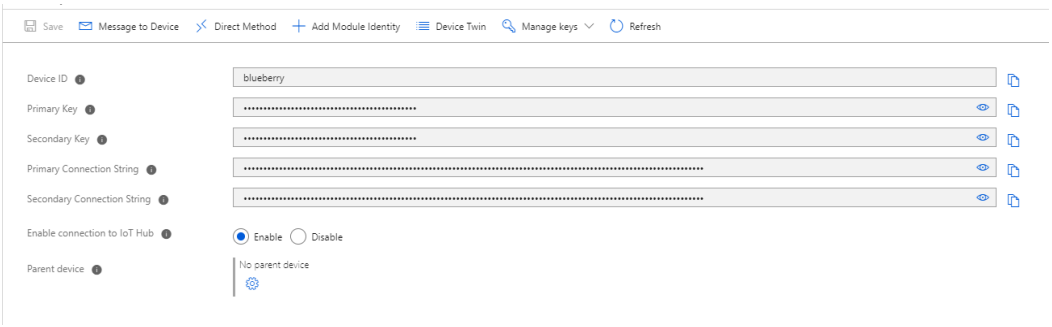

Figure 2: Microsoft Azure IoT Hub device configuration.

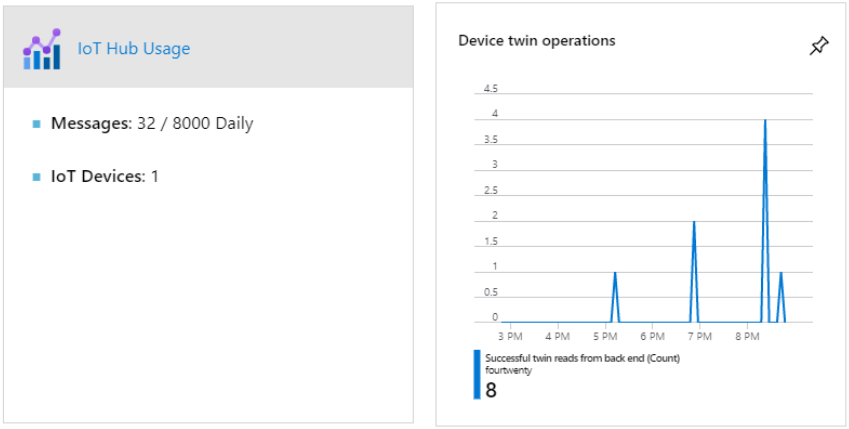

Figure 3: Microsoft Azure IoT Hub usage metrics.