Cloud Based Wireless Sensor Monitoring

Pruthvi Devendrakumar Mistry
Florida Institute of Technology

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation
https://repository.fit.edu/etd/697

This Thesis is brought to you for free and open access by Scholarship Repository @ Florida Tech. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholarship Repository @ Florida Tech. For more information, please contact kheifner@fit.edu.
Cloud Based Wireless Sensor Monitoring

by

Pruthvi Devendrakumar Mistry

A thesis submitted to the College of Engineering and Science of Florida Institute of Technology in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Melbourne, Florida
December, 2018
We the undersigned committee hereby approve the attached thesis, “Cloud Based Wireless Sensor Monitoring” by Pruthvi Devendrakumar Mistry.

Josko Zec, Ph.D.
Associate Professor
Computer Engineering and Sciences
Major Advisor

Carlos Otero, Ph.D.
Associate Professor
Computer Engineering and Sciences
Committee Member

Ming Zhang, Ph.D.
Professor
Aerospace, Physics and Space Sciences
Committee Member

Philip Bernhad, Ph.D.
Head and Professor
Department of Computer Engineering and Sciences
Abstract

Title: Cloud Based Wireless Sensor Monitoring

Author: Pruthvi Devendrakumar Mistry

Major Advisor: Josko Zec, Ph.D.

Internet of Things (IoT) brought a revolutionary change in embedded systems. It offloads the physical work of managing an equipment, appliance or device to a smart and network connected application. IoT makes it possible to monitor and control a huge network of devices remotely.

The aim of this project is to develop a complete framework for smart data acquisition from a wireless sensor network to cloud servers. The framework includes configurable, low power, and small size embedded module that is attached to a sensor. The device reads data from the sensor and forwards them to the cloud server using the wireless interface. The server receives data from the device, configures devices, and creates a dashboard for user interactions. The device firmware can also be updated from the server without requiring any physical contact. The server software written for this framework is highly scalable, so it can be scaled on the cloud to handle many client devices at any location without requiring any changes in the code.
As proof of concept, the data acquisition framework is implemented to constantly monitor radiation intensity. Continuous radiation monitoring can be useful in hospitals and research lab that uses radioactive materials to detect radiation leakage. Beta and Gamma radiation can penetrate living cells and cause cancer, skin burn or organ failure, so it is very critical that radiation is measured from a remote location. The amount of Beta and Gamma radiation is sensed using a Geiger-Muller tube and these data are collected by a very low power embedded device. The collected data is then sent to a server. The server set data retrieval intervals and device sleep cycles.
# Table of Contents

Table of Contents ................................................................................................................. v

List of Figures .................................................................................................................. viii

Acknowledment .................................................................................................................. ix

Chapter 1 Introduction ....................................................................................................... 1

1.1 Cloud System ............................................................................................................. 1

1.1.1 Infrastructure as a Service (IaaS) ..................................................................... 2

1.1.2 Platform as a Service (PaaS) ............................................................................. 2

1.1.3 Software as a Service (SaaS) ............................................................................. 3

1.2 Radiation .................................................................................................................... 3

1.2.1 Ionizing Radiation ............................................................................................. 3

1.2.1.1 Alpha Radiation ............................................................................................ 4

1.2.1.2 Beta Radiation .............................................................................................. 4

1.2.1.3 Gamma Radiation ........................................................................................... 4

Chapter 2 Hardware System Architecture ........................................................................ 5

2.1 Overview .................................................................................................................... 5

2.2 Wireless SoC Module ............................................................................................... 5

2.3 Radiation Sensor ....................................................................................................... 7

2.3.1 Geiger Muller Tube ........................................................................................... 7

2.3.1.1 Working of Geiger-Muller Tube .................................................................... 7

2.3.2 Circuit Board ....................................................................................................... 8

Chapter 3 Software System Architecture ........................................................................ 10
3.1 Embedded Device Software ................................................................. 10
   3.1.1 Configuration Classes ............................................................... 10
   3.1.2 Sensor Data Storage Class ...................................................... 11
   3.1.3 Over the Air Update Handler .................................................. 11
   3.1.4 Interrupt Counter .................................................................. 11
   3.1.5 Main Sequence ....................................................................... 11
      3.1.5.1 Setup Sequence ............................................................... 11
      3.1.5.2 Loop Sequence ............................................................... 12
      3.1.5.3 Wi-Fi Setup .................................................................. 12
      3.1.5.4 Timekeeping ................................................................. 12
      3.1.5.5 Server Synchronization .................................................. 13
      3.1.5.6 Sleep Mode ................................................................ 13
   3.1.6 Compilation Results ............................................................... 15
3.2 Server Software .................................................................................. 15
   3.2.1 Hypertext Transfer Protocol (HTTP) ........................................ 15
   3.2.2 Database Server .................................................................... 16
   3.2.4 Web Server ............................................................................ 17
      3.2.4.1 Collect Readings ............................................................. 17
      3.2.4.2 Device Configurations .................................................. 17
      3.2.4.3 Over the Air Updates .................................................... 18
      3.2.4.4 User Dashboard ............................................................. 19
3.3 Cloud Deployment ............................................................................. 21
   3.3.1 IaaS Deployment ................................................................. 21
   3.3.2 PaaS Deployment ................................................................. 21
   3.3.3 SaaS Deployment of Database Server .................................. 21
Chapter 4 Conclusion and Future Work ................................................. 22
   4.1 Conclusion ................................................................................ 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 Future Work</td>
<td>22</td>
</tr>
<tr>
<td>References</td>
<td>24</td>
</tr>
<tr>
<td>Appendix A ESP8266 Debug Log</td>
<td>25</td>
</tr>
<tr>
<td>Appendix B ESP8266 Source Code</td>
<td>27</td>
</tr>
<tr>
<td>main.ino</td>
<td>27</td>
</tr>
<tr>
<td>config_local.cpp</td>
<td>42</td>
</tr>
<tr>
<td>config_local.h</td>
<td>52</td>
</tr>
<tr>
<td>data_payload.cpp</td>
<td>55</td>
</tr>
<tr>
<td>data_payload.h</td>
<td>60</td>
</tr>
<tr>
<td>ota.cpp</td>
<td>62</td>
</tr>
<tr>
<td>ota.h</td>
<td>63</td>
</tr>
<tr>
<td>utils.cpp</td>
<td>64</td>
</tr>
<tr>
<td>utils.h</td>
<td>68</td>
</tr>
<tr>
<td>Appendix C Server Code</td>
<td>70</td>
</tr>
<tr>
<td>index.js</td>
<td>70</td>
</tr>
<tr>
<td>configPage.ejs</td>
<td>82</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1 - Hardware Architecture ................................................................. 5
Figure 2 - ESP8266 Functional Block Diagram[1] ........................................ 6
Figure 3 - General Arrangement of Geiger-Muller Tube[2] .......................... 7
Figure 4 - Voltage Multiplier Circuit ........................................................... 8
Figure 5 - Simplified detector circuit [2] ..................................................... 9
Figure 6 – Wi-Fi Setup Process ................................................................. 13
Figure 7 - Data Send Process ................................................................. 14
Figure 8 - Configuration Retrieval Process ............................................... 14
Figure 9 - POST Sensor Data to Server Flow ........................................ 18
Figure 10 - GET Config Flow ................................................................. 18
Figure 11 - OTA Update Flow ................................................................. 19
Figure 12 - Change Configurations from User Dashboard Flow ............. 19
Figure 13 - User Dashboard Flow .......................................................... 20
Figure 14 - User Dashboard Screenshot ................................................... 20
Figure 15 - Debug Logs after Setup Sequence ........................................ 25
Figure 16 - Debug Logs after Loop iteration ........................................... 26
Acknowledgment

I would like to thank my thesis advisor Dr. Josko Zec for inspiring me to work on this thesis. Completion of the thesis would not be possible without his constant help and support. He encouraged and guided me while I was working on the thesis.

I am grateful for the thesis defense committee members Dr. Carlos Otero and Dr. Ming Zhang. Without their expert insight on the topic the thesis could not succeeded.

I would like to express my gratitude to my parents for always supporting and encouraging me throughout my study.
Chapter 1
Introduction

1.1 Cloud System

A cloud is like a one-stop shop for all the infrastructure required to deploy any application. Cloud can deliver database, computer servers, software services and more. All these services are provided on the on-demand basis. It has changed the traditional way of thinking about computing resources by replacing the rigid server model to a service-based model. The traditional server model is mainly focused on providing a generic server and network infrastructure. The user must manage server load balancing, deployment cycle management, and many other operational tasks in addition to develop the main application. One of the most critical issue with traditional server models is that the user must create a failsafe redundant software to handle server failures. Thus, there is a lack of abstraction and scalability.

The cloud model was developed to provide a variable level of abstraction and high scalability. Scalability is the ability to expand the system whenever required. Cloud provides scalable services for load balancing, database, server replication and more.

Abstraction means removing granular details of the system in order to focus on the actual functionality of an application. It simplifies how the server is seen from the developers’ perspective and helps them focus on the overall application rather than nitty gritty parts of managing server. Cloud abstracts the server management from the user, so the application developer does not have to worry about the managing servers.
Based on the level of abstraction provided by the cloud there are three types of cloud services,

1.1.1 Infrastructure as a Service (IaaS)

This type of cloud service provides the whole server infrastructure to the user. The server infrastructure includes virtual computers, network routing, network load balancing, and firewalls. It is up to users how they want to utilize those resources. It provides fine grain control over the resources. The resources might be virtual on the cloud, but this detail is abstracted from the user. For example, the physical server computer used to host the cloud might have multiple processors, very high amount of memory and hard drive, however, if the requested resources are low then the cloud virtualizes those physical resources into a small virtual resource and the virtualization is transparent to the user. All these resources are virtualized independently, so they can be scaled independently. For example, it is possible to upgrade the CPU, memory, or hard drive on the server without changing the others. It is just like changing physical parts of a computer.

1.1.2 Platform as a Service (PaaS)

It abstracts the server infrastructure from the user. That means the user does not require to have any knowledge of the server computers that will be running the application. It is very easy to deploy on this type of cloud services. It allows the developers to focus on developing apps and not worry about managing them. Most of the PaaS cloud integrates a Continuous Deployment, Continuous Testing, and Continuous Integration service which speeds up the development cycle.

The PaaS uses containers which acts as a resource to run the application. A container is an abstraction layer above the operating systems. The container creates an image which includes the app and all the dependencies required to run the app. The images are self-contained, so they can run on any physical or virtual platform without having to configure it differently. They also provide a layer of isolation between multiple apps running on the same server, as well as the server operating system.
1.1.3 Software as a Service (SaaS)

It is a cloud-based software provided as a service. It provides flexible on-demand use of the corporate-owned software. For example, Google provides Machine Learning services that we can use to process data. The software and algorithms are owned by Google and they provide it as a paid service.

1.2 Radiation

Radiation is the release of energy and matter from the matter in the form of waves or high-speed particles which can travel in space or through any source of medium. The matter is made up of atoms. Atom is made up of the nucleus – the central part which is a cluster of positive charge carrying proton and electrically neutral neutron and an orbit surrounding it of negatively charged electrons. These forces inside atom work toward balancing and stabilizing itself.

When there is excessive protons or neutrons in the nuclei the atom becomes unstable because the internal forces are unable to keep the nuclei bonded together. A material that has unstable nuclei is called as radioactive material. The unstable nuclei try to achieve a stable state by throwing out the excess protons or neutrons. This process is known as radioactive decay which can be in form of alpha, beta or gamma radiation.

1.2.1 Ionizing Radiation

Alpha, Beta and Gamma radiations are also known as ionizing radiation because they carry enough energy to ionize atoms and molecules. They can cause cancer or organ failure. They also have several beneficial uses such as radiotherapy for cancer treatment, but it can be harmful if not used under a controlled environment.
1.2.1.1 Alpha Radiation

The particles of alpha radiation resemble a positively charged particle emitted spontaneously from the nuclei of some radioactive elements. It is identical to a helium nucleus - the mass number of 4 and an electrostatic charge of +2. It has short range (few centimeters in air) and low penetrating power at its usual velocity. It cannot penetrate the even outer layer of dead skin cells. Hence, no damage is causing to live tissue. However, it can be dangerous if alpha-emitting isotope is inhaled or swallowed. These particles can be blocked by a paper sheet.

1.2.1.2 Beta Radiation

There are two types of beta radiations. Beta-minus ($\beta^-$) which is an electron and Beta-plus ($\beta^+$) which is a positron. It has more penetrating power than alpha particles as it is lighter than alpha particles. As a result, it can travel a couple of feet in the air and can penetrate the skin. Despite its abilities, a thin sheet of metal, plastic or wood can stop them.

1.2.1.3 Gamma Radiation

The gamma radiation consists of photons with a wavelength even smaller than X-rays, thus making it possible for it to travel great distances. Alpha and beta particles, both have mass and electric charge. Whereas, Gamma radiation consists photons which have neither electric charge nor mass. As a result, it can penetrate further through matter compared to alpha and beta particles. Gamma rays can be blocked by a sufficiently thick or dense material layer.
Chapter 2
Hardware System Architecture

2.1 Overview

The hardware architecture of this project consists of a wireless system on a chip (SoC) module and a Geiger-Muller radiation sensor board. The SoC processes the signals from the sensor and sends them to a server through a Wi-Fi interface as shown in Figure 1. The SoC module can be configured in many ways from the server.

![Figure 1 - Hardware Architecture](image)

2.2 Wireless SoC Module

The wireless SoC is the control part of the system which collects data from a sensor, connects to appropriate Wi-Fi, sends collected data to the server, and receives
configurations or firmware updates from the server. To facilitate these functionalities the SoC module must have a Wi-Fi interface, low power requirement, and enough persistent storage for offline data storage.

The ESP8266 is an ultra-low power SoC with built-in Wi-Fi interface which has all the required features. An ESP8266 based NodeMCU development board was used to implement this project. It has integrated TCP/IP protocol stack and FCC, Wi-Fi Alliance certified Wi-Fi 2.4GHz WPA2. The integrated microprocessor in ESP8266 is a Tensilica L106 32bit microcontroller which runs at maximum CPU frequency of 160MHz. It has 80kB of SRAM and supports up to 16MB of external flash memory [1]. It runs at 3.3V and typical current usage is between 50mA to 170mA.

It is compact in size and outputs reliable performance making it capable to perform either as a standalone application or a slave to a host microcontroller unit. In this project, it was used as a standalone device. NodeMCU is an open source platform based on ESP8266 for developing internet of thing (IoT) devices. As a result, work for hardware configuration and manipulation reduced to a greater extent. The functional block diagram is as shown in Figure 2.

![ESP8266 Functional Block Diagram](image)

**Figure 2 - ESP8266 Functional Block Diagram[1]**
2.3 Radiation Sensor

2.3.1 Geiger Muller Tube

In this project, the J305β Geiger-Muller tube is chosen, as it can detect beta and gamma particle radiation which was the focus of this project. The tube is a type of particle detector which detects radiation by detecting ionization of the low-pressure gas. The tube is designed in such a way that a window exists at one end through which ionizing radiation can penetrate easily. While the other end has electrical connectors. It is as shown in Figure 3. This type of window is also known as end-window.

![General Arrangement of Geiger-Muller Tube](image)

**Figure 3 - General Arrangement of Geiger-Muller Tube[2]**

2.3.1.1 Working of Geiger-Muller Tube

When radiation particles like alpha, beta or gamma enters the tube, it ionize the gas inside the tube. When sufficient high electric field is applied to the tube, ions and electrons are able to separate fully. These generated ions and electrons are then attracted to electrodes as the ionized gas acts as a conductor. This flow of electron creates a current pulse in the tube which later is sensed by associated voltage in the signal conditioning circuit. Whenever a particle is detected, a pulse of current is generated.

The Geiger-Muller tubes gives output as a number of pulses thus the radiation intensity and not an energy value. More the pulses more the amount of radiation. However,
a number of pulses also depends on the tube. A standard unit to measure radiation is microsieverts per hour. We can convert the pulse count to microsieverts per hour by multiplying it by a constant conversion factor of the tube. \((\text{Counts per minute}) \times \text{(Conversion factor)} = \mu\text{Sv/h},\) where the conversion factor for the J305β tube is 0.008120370.

### 2.3.2 Circuit Board

The circuit controlling the Geiger-Muller tube has two main circuits. The first circuit generates high voltage fed to the tube and another circuit is output conditioning circuit.

The Geiger-Muller tube requires very high power approximately 370V DC. This voltage can be generated using an oscillator connected to voltage multiplier. A general circuit of voltage multiplier is shown in Figure 4.

![Figure 4 - Voltage Multiplier Circuit](image)

The other part of the circuit is the signal handling circuit. It is a simple RC filter followed by a transistor-based signal isolation circuit. The RC filter removes the high voltage spikes from the tube to protect the transistor circuit. The transistor provides an isolation between the 5V output of the circuit and the high voltage. The circuit for signal handling is shown in Figure 5.
Figure 5 - Simplified detector circuit [2]
Chapter 3
Software System Architecture

3.1 Embedded Device Software

The microcontroller unit (MCU) of ESP8266 can be programmed in many ways, including but not limited to ESP SDK based C/C++, Lua script on top of NodeMCU firmware, Python script on top of MicroPython runtime, Arduino based C++. In this project code for ESP8266 is completely written in C++ on top of Arduino standard libraries. This method was chosen because of the high amount of community support for the Arduino platform, easiness of programming with Arduino libraries, and high performance of C++ code. The code could have been optimized further using native ESP SDK based C/C++ method, however, it would require strong familiarity with the SDK and long development time.

The device code implements the client-side Application Program Interface (API) of the server. Each device has its unique identifier known as Token. The server identifies the device based on this Token value. The Token value is a fixed string at the compile time. The source code for ESP8266 is in Appendix B ESP8266 Source Code. It is divided into different classes. Following are the major parts of the code.

3.1.1 Configuration Classes

There are two classes to manage configuration data, config_usr and wifi_config. config_usr class manages data retrieval time and sleep time configuration, and wifi_config class manages Wi-Fi credentials. Both classes implement methods to store configuration
on persistent storage, load configuration from the persistent storage, and fetch configuration from the server.

3.1.2 Sensor Data Storage Class

This class manages the sensor reading data. It manages the pulse count, reading time interval, and timestamp. These are three unsigned 32bit numbers. The count per minute (CPM) value is not calculated on the MCU because floating point operations on the MCU could be very costly in terms of power, time, and memory. This class implements a method to store (append) data to persistent storage, read data from persistent storage, and send data to the server.

3.1.3 Over the Air Update Handler

The over the air update handler uses the httpUpdate library. It has functions to check for update on the server and perform an OTA update. These functions act as a custom wrapper around the httpUpdate library [4].

3.1.4 Interrupt Counter

The interrupt counter counts at every external interrupt from the radiation sensor. The interrupt signal from the sensor is a low pulse. The interrupt is set as a rising edge sensitive interrupt, i.e. the interrupt is generated at low to high pulse on the pin.

3.1.5 Main Sequence

This is the main algorithm of the code. Just like any Arduino code, it has two main sequences, setup, and loop. The setup is a function which called only once when reset. The loop is a function which called in an infinite loop after setup.

3.1.5.1 Setup Sequence

The setup function initializes the serial interface, initialize persistent storage, setup interrupts and interrupt service routines, set up Wi-Fi connection, setup network timing protocol (NTP) client, does initial server synchronization. Appendix A ESP8266 Debug Log shows debug log after setup sequence. Sensitive data are hidden.
3.1.5.2 Loop Sequence

The loop function reads sensor data according to the configuration, then setup Wi-Fi if not connected, updates NTP client, does server synchronization, and sleeps according to the configuration. Loop keeps repeating these actions in sequence. Appendix A ESP8266 Debug Log shows debug logs after a loop iteration.

3.1.5.3 Wi-Fi Setup

The program initially tries to connect using Wi-Fi credentials stored in a configuration class. On failure, it tries to connect using a list of default Wi-Fi credentials stored at compile time. On its failure and if a specific pin (for this project, pin D4) is connected to the ground, it creates a Wi-Fi hotspot where we could connect and enter Wi-Fi credentials. However, the credentials sent through hotspot are not stored and should be only used for first time setup. Figure 6 shows the Wi-Fi setup process.

3.1.5.4 Timekeeping

Timekeeping is a critical part of the code. When device gathers data while it is offline, it needs to keep track of time when data was collected. The ESP8266 does not have an internal real-time clock (RTC), to compensate this the best way to accurate timing would be to use an external real-time clock (RTC) with the separate power supply on it. However, it would increase the cost, power, and size of the device. It is not critical to keep very accurate timing; inaccuracy of a few seconds would not make any difference.

Considering these, network time protocol (NTP) is a good solution for a time source. NTP is standard internet protocol used for timekeeping on computers as well as embedded devices. The program gets time from NTP server when it starts. Once the NTP client gets time from the server, the time on the device is continuously updated through an internal timer. So, even if the device gets offline, NTP client keep track of time through the timer. The NTP client keeps synchronizing time from the server at every 5 minutes.
### 3.1.5.5 Server Synchronization

Server synchronization is a sequence that exchanges data, configuration, and OTA with the server. It tries to send locally stored sensor reading data (if any) as well as currently read data to the server, on failure, it stores (appends) current data to persistent storage. If success, it removes all stored reading data from persistent storage as shown in Figure 7. Then it tries to fetch configuration from the server, on success, it updates those configurations on persistent storage as shown in Figure 8. It also calls the OTA sequence.

### 3.1.5.6 Sleep Mode

The ESP8266 have a built-in automatic sleep mode. It is automatically started when the code is doing nothing. During sleep mode, Wi-Fi radio is shut down to reduce power usage. The current used during sleep mode is about 15mA [1].

---

**Figure 6 – Wi-Fi Setup Process**

Connect using `wifi_config`

- **Yes, Connected?**
  - **Yes, Connected:**
    - Start WiFi Management Hotspot to get temporary WiFi Credential
    - Connect using default WiFi set
  - **No:**
    - WiFi Connected
- **No:**
  - **Connected?**
    - Is pin D4 grounded?
    - Yes, Connected: WiFi Connected
    - No: WiFi Not Connected

[1] Source reference for current consumption during sleep mode.
Figure 7 - Data Send Process

Figure 8 - Configuration Retrieval Process
3.1.6 Compilation Results

The code is well compressed and optimized for the memory and flash storage. Compiled binary of the program takes 334kB of flash storage which is only 32% of available program storage space. This gives plenty of room for persistent storage as well as future improvements. The program uses about 38kB of memory for global variables (heap), leaving 42kB for the dynamic variable (stack) which is enough to facilitate complex code structures. These results are very useful to understand if this code can be implemented for an even lower power device.

3.2 Server Software

The server software is written in NodeJS which is a server-side JavaScript runtime. The NodeJS uses non-blocking I/O and event loops to parallelize the program. The non-blocking, I/O means that I/O operations such as reading or writing network socket or file do not halt rest of the programs. Instead of blocking the main sequence of code, we use callback functions, which are called when an I/O event is raised.

Thanks to huge community support, it is very fast and easy to build complex server programs. The main server is an HTTP server, and it is mainly written using ExpressJS [5] library which provides basic HTTP server facilities. There is also a database server of MongoDB.

The code written for the server side of this project is optimized to be able to deploy on PaaS cloud. A single instance of the server can handle several such devices. However, we can scale up easily to handle millions of user and device sessions. It does not require us to modify the code at all. The cloud can be scaled to multiple instances across different geographical locations to provide low latency service.

3.2.1 Hypertext Transfer Protocol (HTTP)

HTTP is an application layer protocol which is the foundation of any data exchange over the web. HTTP is a client-server based protocol so it works on top of the TCP/IP [6]. Only a client can send a request and server would respond. Common request
method for HTTP is, GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, and TRACE. We used HTTP GET and POST methods for this project. GET method is used to retrieve data from the server. In this implementation, it is used to fetch configuration data from the server to embedded devices. POST method is used to send large chunks of data to the server. In this implementation, it is used to send the collected data from the device to the server. Structure of HTTP request packet is,

```
<METHOD> /path/on/server HTTP/1.1
<HEADER> : <HEADER VALUE>
<empty line>
<data>
```

The headers are fundamental parts of any HTTP packet which contain metadata. Metadata could be the size of the content being sent, type of content, accepted language, browser type etc. There are some standard headers defined by the Internet Engineering Task Force (IETF). We can also add custom headers, but we must make sure these headers are properly handled by the server as well as the client. In this software implementation, we are using a custom header “Token” which uniquely identifies an embedded device. Thanks to this header, a single server can handle a huge number of embedded devices. The data can be in either plain text or binary format and its type can be specified by the “Content-type” header.

### 3.2.2 Database Server

A MongoDB server was used to manage all the collected sensor data, configuration data, and over the air update versions. MongoDB is a NoSQL type server [7], meaning it stores data as a collection of JavaScript Object Notation (JSON) format rather than conventional tables. The JSON format is a key-value pair of data. Key is always a string. A value can be a string, number, array or a JSON object. The pair of Key and Value specifies the JSON format. Following is an example of a JSON object.

```
{
    "str" : "value",
```
The structure of the JSON object is not forced by the database server, it is handled by the database client, in this case, the database client is the web server. This type of database is suitable for this project because the format of data is not rigid, so we can rapidly develop and reuse the same database for a different purpose, i.e. if we want to use the same database for collecting data on the different type of sensor, we can do it with minimum change in the database structure. Also, the JSON format is native to JavaScript which allows easy interaction with the web server written in JavaScript.

3.2.4 Web Server

The server has 4 basic functions each function can be used for many devices. It handles requests from the embedded device and shows a dashboard for user interactions.

3.2.4.1 Collect Readings

The device sends a POST request along with collected data to the web server in Comma Separated Vector (csv) format. The server stores that data into the database. Each data entry has a pulse count value, timestamp value, and a duration value. The timestamp is a 32bit unsigned number which represents seconds since January 1st, 1970 midnight at London. It is a standard format for keeping time. The sequence diagram for this function is as shown in Figure 9.

3.2.4.2 Device Configurations

Configurations could be data retrieval time, sleep time, and Wi-Fi credentials. These configurations are stored on a database and whenever asked by the device, they are sent to them. All the configurations are transferred in JSON format. The server stores
separate configurations for each Token value. Configurations are stored on the database server. The sequence diagram is as shown in Figure 10.

Figure 9 - POST Sensor Data to Server Flow

Figure 10 - GET Config Flow

3.2.4.3 Over the Air Updates

The server allows the user to upload the binary file. This binary can be flashed on to the device remotely. This functionality is very useful as it allows easily the firmware updates without getting device offline. Figure 11 shows the Over the Air update flow.
3.2.4.4 User Dashboard

The dashboard shows all the data collected from a device, allows the user to set configurations, and do an over the air update. The flow to change configurations from user dashboard is as shown in Figure 12. Each device with a unique Token id has its own dashboard. The flow which makes possible to see dashboard for each individual device is as shown in Figure 13. Screenshot of the dashboard is as shown in Figure 14.

![Diagram of OTA Update Flow](image)

**Figure 11 - OTA Update Flow**

![Diagram of Change Configurations from User Dashboard Flow](image)

**Figure 12 - Change Configurations from User Dashboard Flow**
Figure 13 - User Dashboard Flow

Basic Config for Device Id: testtoken

Basic Config

<table>
<thead>
<tr>
<th>device id</th>
<th>configId</th>
<th>data retrieval time in ms</th>
<th>sleep time in ms</th>
</tr>
</thead>
</table>

WiFi Config

<table>
<thead>
<tr>
<th>device id</th>
<th>ssid</th>
<th>passwd</th>
</tr>
</thead>
</table>

OTA Update

Firmware Bin File: Choose File | No file chosen

Submit

Set OTA Version

<table>
<thead>
<tr>
<th>configId</th>
<th>data Retrieval TimeMs</th>
<th>sleep TimeMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>123213</td>
<td>11000</td>
<td>1000</td>
</tr>
</tbody>
</table>

password (sha256)
pnthiv's iPhone cb68499aee34ffde26e3a40751616824b4e0f516d11d9853a5e4183906e27bd4

uSv: Constant:

Update Constant

Clear Data

Download Content

<table>
<thead>
<tr>
<th>timestamp</th>
<th>count</th>
<th>counting time</th>
<th>CPM</th>
<th>uSv/h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1543117369</td>
<td>0</td>
<td>11002</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1543117355</td>
<td>2</td>
<td>11000</td>
<td>10.909090909090908</td>
<td>0.08727272727272728</td>
</tr>
<tr>
<td>1543117343</td>
<td>3</td>
<td>11002</td>
<td>16.3606616978313</td>
<td>0.13088529358298492</td>
</tr>
<tr>
<td>1543117330</td>
<td>3</td>
<td>11001</td>
<td>16.36214889555495</td>
<td>0.1308971911644396</td>
</tr>
<tr>
<td>1543117317</td>
<td>0</td>
<td>11001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1543117304</td>
<td>0</td>
<td>11001</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 14 - User Dashboard Screenshot
3.3 Cloud Deployment

Scalability of the application was tested by deploying the application on different types of cloud servers. We used IaaS cloud provided by Google Cloud Platform and PaaS cloud provided by Heroku to deploy the web server and SaaS cloud provided by MongoDB Atlas for MongoDB database server. The device or user can connect to either of the endpoints (server address) and there will be no change in functionality.

3.3.1 IaaS Deployment

To deploy the web server on the IaaS cloud, we had to create a compute resource. Compute resource provides a virtual machine. We had to upload the server code on the server. We also had to install the NodeJS runtime on the server to run the code. NodeJS has a dependency manager called Node Package Manager (npm). The package manager was used to install all the code dependencies on the server. We had to set up networking on the cloud to expose the server to the public internet. All of these we had to do manually. Code failure and server reset were not managed by the cloud. It took a long time to figure out and run those steps. However, it provides a greater control over the deployment.

3.3.2 PaaS Deployment

To deploy the web server on the PaaS cloud, we just had to upload the code to the server. The PaaS cloud setup all dependencies and runs the code. Runtime installation, network setup, and code failure were handled by the cloud. Changing code is also simple because we did not have to reboot the server ourselves it was automatically by the cloud. We could scale the server by just selecting how much computing and network resources we require. Allocation of the resources and load balancing was completely transparent.

3.3.3 SaaS Deployment of Database Server

We can create the database server using SaaS cloud with a click of a button and it can connect it our web server without changing the source code. However, doing the same with IaaS would require us to manage the database software and all the networking required for it.
Chapter 4
Conclusion and Future Work

4.1 Conclusion

The framework for data acquisition was successfully designed and implemented for monitoring radiation. The embedded device was able to collect sensor data and send them over a wireless network. The device is smart enough to handle conditions when the wireless network is not available. A server was written that could control and monitor the sensor device. The database server on the back end of the main server is flexible enough to handle any changes in data scheme which allows rapid expansion. The server also creates a dashboard for user interactions. The device firmware could be updated through the web server. This framework can be very useful as a basis to develop large-scale IoT applications. The small code size allows using smaller, cheaper, and low power microcontrollers which are a big advantage as an IoT solution. This project was able to leverage advantages of all three types of cloud services by using each for a specific purpose.

4.2 Future Work

This framework can be used for different type of sensor monitoring with minor changes in code. A single embedded device could be used to collect data from multiple sensors.

The current implementation uses unsecured HTTP protocol for interactions between the server and the device. It can be secured by using the HTTPS protocol. A low overhead protocol like MQTT can be used.
Load balancing could be used between the deployments to distribute computation and network load.

We can use Low Power Wide Area Network protocols like LTE-M, NB-IoT, LoRa, and SigFox for wireless connectivity to the server. However, the hardware required for these would increase the cost of the solution ten folds.

User dashboard can be improved to process collected data and show statistics. A machine learning model can be used to predict future data or correlated collected sensor data with some other data set.
References


Appendix A
ESP8266 Debug Log

INFO: Not formatting SPIFFS
INFO: SPIFFS initialized. Space: 1044/95714 bytes
WARNING: SSID not found! Aborting connection try with following credentials.
INFO: SSID authenticated! PASSphrase: [redacted]
INFO: Failing back to device default WiFi credentials.
INFO: Connected to WiFi.
Mode: STA
PHY mode: N
Channel: 11
IP id: 0
Status: 5
Auto connect: 1
SSID: [redacted]
Passphrase: [redacted]
RSSID set: 0
INFO: Setting up Network Time Protocol (NTP) Client
INFO: Syncing with NTP
INFO: Current Timestamp: 1562307772.0 Formatted Time: 11:28:43
INFO: Getting Config from Cloud
WARNING: data payload Can't Store SPIFFS
INFO: Checking FOR OTA UPDATE

HTTP GET URL: https://15.127.180.122/ota_check

HTTP GET RESPONSE: 1
OTA UPDATE RESPONSE:
INFO: OTA UPDATE AVAILABLE

HTTP GET URL: https://15.127.180.122/wifi_config

HTTP GET RESPONSE: "response_code":1,"error":null,"passw":"[redacted]"
INFO: wifi_config from cloud success
INFO: wifi_config stored to SPIFFS

HTTP GET URL: https://15.127.180.122/config_user

HTTP GET RESPONSE: "response_code":1,"config":null,"dataRetriwalTime":10000,"sleepTimeMin":10000
INFO: Success got config_user from cloud.
PRINT TO FILE
"config":null,"dataRetriwalTime":10000,"sleepTimeMin":10000
FILE OPENED
INFO: config_user stored to SPIFFS success
INFO: Cloud Sync Sequence Success.

Figure 15 - Debug Logs after Setup Sequence
INFO: In main loop.
INFO: Retrieving Data with config
INFO: Reading...

Figure 16 - Debug Logs after Loop iteration

HTTP GET URL: https://google.com

HTTP GET RESPONSE: HTML

HTTP POST URL: https://35.227.180.122/data_payload

HTTP POST RESPONSE: {
  "response_code": 11
}
INFO: Success need to cloud.
INFO: CHECKING FOR OTA UPDATE

HTTP GET URL: https://35.227.180.122/ota_check

HTTP GET RESPONSE:
OTA UPDATE RESPONSE:
INFO: NO OTA UPDATE AVAILABLE

HTTP GET URL: https://35.227.180.122/dfsi_config

HTTP GET RESPONSE: {
  "response_code": 1,
  "ssid": "asdfsdfasdf",
  "passw": "Ffgjhnmmmmmmmmm"
}
INFO: wifi_config from cloud success
INFO: wifi_config stored to SPIFFS

HTTP GET URL: https://35.227.180.122/config_user

HTTP GET RESPONSE: {
  "response_code": 1,
  "configId": "123213",
  "dataRetrivalTime": "156999", "sleepTime": "10000"
}
INFO: Success get config_user from cloud.
PRINT TO FILE
{"configId": "123213", "dataRetrivalTime": "156999", "sleepTime": "10000"}
INFO: config_user store to SPIFFS success
INFO: Printing all data_payload content...
Appendix B
ESP8266 Source Code

main.ino

#include "config_local.h"
#include "data_payload.h"
#include "utils.h"
#include "ota.h"
#include "FS.h"

#include <NTPClient.h>
#include <ArduinoJson.h>
#include <DNSServer.h>

#if defined(ESP8266)
#warning "COMPILING FOR ESP8266"
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <ESP8266HTTPClient.h>
#include <WiFiManager.h>
#include <ESP8266httpUpdate.h>
#else
#warning "NOT COMPILING FOR ESP8266"
#include <WiFi.h>
#include <HTTPClient.h>
#include <WebServer.h>
#include <SPIFFS.h>
#include <Update.h>
#endif

#endif
// Debug Printing Setting
#define PRINT_DEBUG
#define PRINT_DEBUG_SENSITIVE

#ifdef PRINT_DEBUG
    #define debugLogF(a) Serial.println(F(a))
    #define debugLog(a) Serial.println(a)
#else
    #define debugLogF(a)
    #define debugLog(a)
#endif

#ifdef PRINT_DEBUG_SENSITIVE
    #define debugLogSensitive(a) debugLog(a)
#else
    #define debugLogSensitive(a)
#endif

#define DEFAULT_WIFI_COUNT 3

// Globals
volatile unsigned long interruptCounter;
bool spiffsStatus;
config_usr cu;
data_payload dp;
wifi_config wc;

const int currentVersion = 0;

// Default Constants
#if defined(ESP8266)
const unsigned int defaultInterruptPin = D6;
const unsigned int SPIFFSFormatPin = D3;
const unsigned int defaultWifiManagerPin = D4;
#else
const unsigned int defaultInterruptPin = 6;
const unsigned int SPIFFSFormatPin = 7;
#endif
const char* defaultWiFiManagerSSID = "WiFiManagementHotSpot";
const char* defaultWiFiManagerPasswd = "thisisatestpassword";
const char* defaultWiFiSSID[] =
{"ssid1","ssid2","ssid3","ssid4"};
const char* defaultWiFiPasswd[] =
{"passwd1","passwd2","passwd3","passwd4"};
unsigned short defaultWiFiListLength = 4;
const unsigned long defaultNTPUpdateTimeS = 300;

// Initialize NTP
WiFiUDP ntpUDP;
NTPClient ntpClient(ntpUDP);

// Prototypes
wl_status_t wifiConnect(String ssid, String passwd, unsigned long timeout=20000);
wl_status_t wifiConnect(const char* ssid, const char* passwd, unsigned long timeout=20000);
wl_status_t wifiConnectList(const char* ssidList[], const char* passwdList[], int listSize, unsigned long timeout=2000);
wl_status_t wifiSetupSequence();
bool isSSIDAvailable(String ssid);
bool initWifi();

bool initSPIFFS(bool format);
void interruptHandler();
bool cloudSyncSequence();
void doSleep();

void setup() {
// Initialize interrupt counter
    interruptCounter = 0;
    spiffsStatus = false;

// Initialize Serial
    Serial.begin(115200);
    debugLogF("INFO: Serial Initialized");

    pinMode(defaultWifiManagerPin, INPUT_PULLUP);
// Initialize SPIFFS
// Set to true to format SPIFFS for first use.
    pinMode(SPIFFSFormatPin, INPUT_PULLUP);

    if(digitalRead(SPIFFSFormatPin)){
        debugLogF("INFO: Not formatting SPIFFS");
        if(!initSPIFFS(false)){
            debugLogF("WARNING: Unable to initialize SPIFFS");
            spiffsStatus = false;
        }else{
            spiffsStatus = true;
            #if defined(ESP8266) && defined(PRINT_DEBUG)
                FSInfo fsi;
                SPIFFS.info(fsi);
                debugLog(String("INFO: SPIFFS Initialized. Space:
                    ") + fsi.usedBytes + "/" + fsi.totalBytes + " bytes");
            #else
                debugLog(String("INFO: SPIFFS Initialized. Space:
                    ") + SPIFFS.usedBytes() + "/" + SPIFFS.totalBytes() + " bytes");
            #endif
        }
    }
}
#endif
}
}
else{
    debugLogF("INFO: Formatting SPIFFS");
    if(!initSPIFFS(true)){
        debugLogF("WARNING: Unable to initialize SPIFFS");
        spiffsStatus = false;
    }else{
        #if defined(ESP8266) && defined(PRINT_DEBUG)
            FSInfo fsi;
            SPIFFS.info(fsi);
            debugLog(String("INFO: SPIFFS Initialized. Space:
            ") + fsi.usedBytes + "/" + fsi.totalBytes + " bytes");
        #else
            debugLog(String("INFO: SPIFFS Initialized. Space:
            ") + SPIFFS.usedBytes() + "/" + SPIFFS.totalBytes() + " bytes");
        #endif
        spiffsStatus = true;
    }
}
// WiFi Initialization Sequence
if(!initWifi()){  
    debugLogF("WARNING: Unable to initialize WiFi module to WIFI_STA mode.");
}
if(wifiSetupSequence() != WL_CONNECTED){
    debugLogF("WARNING: Unable to setup WiFi after setup sequence.");
}

// Setup NTP ntpClient
    debugLogF("INFO: Setting up Network Time Protocol (NTP) Client");
    ntpClient.begin();
// Update from NTP every 5 minutes.
tcpClient.setUpdateInterval(defaultNTPUpdateTimeS);

if(WiFi.status() == WL_CONNECTED){
    debugLogF("INFO: Syncing with NTP");
    tcpClient.forceUpdate();
    debugLog(String("INFO: Current Timestamp:
")+tcpClient.getEpochTime()+" Formatted Time:
"+tcpClient.getFormattedTime());
}

ddebugLogF("INFO: Getting Config from Cloud");
if(WiFi.status() != WL_CONNECTED){
    debugLogF("WARNING: WiFi not connected. Skipping Cloud Sync Sequence.");
} else{
    if(!cloudSyncSequence()){
        debugLogF("WARNING: Cloud Sync Sequence Failed.");
    } else{
        debugLogF("INFO: Cloud Sync Sequence Sucess.");
    }
}

attachInterrupt(defaultInterruptPin,interruptHandler,RISING);

void loop() {
    debugLogF("INFO: In main loop.");

    unsigned long dataRetrivalTime = cu.getDataRetrivalTimeMs();
    debugLogF("INFO: Retriving Data with config: ");

    tcpClient.update();
unsigned long initTimeStamp = ntpClient.getEpochTime();
unsigned long initInterruptCount;
unsigned long endTimeMillis;
unsigned long initTimeMillis;
initTimeMillis = millis();
interruptCounter = 0;
initInterruptCount = interruptCounter;
// DON'T SET TO 0
debugLog("INFO: Reading...");
while(millis() - initTimeMillis < dataRetrivalTime){
    Serial.print(F("."));
    delay(1000);
}

unsigned long endInterruptCount = (unsigned long)( (long long)interruptCounter - (long long)initInterruptCount );
d.debugLog("INFO: Count:"+String(endInterruptCount));

endTimeMillis = millis();
d.debugLog("INFO: endTimeMillis:" + String(endTimeMillis));

ntpClient.update();
d.debugLogF("INFO: Count Ended at ");
d.debugLog(ntpClient.getFormattedTime());

d.debugLogF("INFO: Appending Data");

unsigned long timeMillis = (unsigned long)((long long)endTimeMillis-(long long)initTimeMillis);
d.debugLog("INFO: Writing "+ String(endInterruptCount) + "," + String(timeMillis) + "," + String(initTimeStamp));
dp.append_data(endInterruptCount, timeMillis, initTimeStamp);

dbgLogF("INFO: Setting Up WiFi Again...");
#if defined(PRINT_DEBUG)
    WiFi.printDiag(Serial);
#endif

if(WiFi.status() != WL_CONNECTED){
    if(wifiSetupSequence() != WL_CONNECTED ){
        dbgLogF("WARNING: WiFi Setup Failed");
    }else{
        dbgLogF("INFO: WiFi Setup Success");
    }
}

dbgLogF("INFO: Trying Cloud Sync");
// Test Connection
#if defined(PRINT_DEBUG)
    String r;
    httpGET("http://google.com",r);
    dbgLog(r);
#endif

cloudSyncSequence();

#if defined(PRINT_DEBUG)
    String jsonhacktemp;
    dp._to_json_hack(jsonhacktemp);
    dbgLog(String("INFO: Printing all data_payload content") + jsonhacktemp);
#endif
doSleep();
void doSleep()
{
    unsigned long a = millis();
    while(millis() - a > cu.getSleepTimeMs()){
        delay(1000);
    }
}

bool cloudSyncSequence(){
    // Send Data
    //  bool loadSpiffsWorked=false;
    // Store to SPIFFS
    if(!dp.storeSpiffs()){
        debugLogF("WARNING: data_payload Can't Store SPIFFS");
        //debugLogF("WARNING: CAN'T STORE CAN'T SEND! CODE SUCKS!!!!

    Use vector or learn memory mgt.");
    }else{
        debugLogF("INFO: data_payload SPIFFS stored");
    }
    // Send to cloud
    if(dp.isStored()){
        if(!dp.sendToCloud()){
            debugLogF("WARNING: Can't send to cloud!");
        }else{
            debugLogF("INFO: Sucess send to cloud!");
            dp.rmSpiffs();
        }
    }else{
        debugLogF("WARNING: No stored no send to cloud. sorry code
doesn't want seg_faults");
    }
}
// Ask for new configs  
// Get Latest Config (and Firmware too) from Cloud. 

// OTA

dbgl("INFO:CHECKING FOR OTA UPDATE");
if(checkForUpdate()){
    dbgl("INFO: DOING OTA UPDATE");
    if(doUpdateOTA()){
        dbgl("INFO:OTA UPDATE SUCESS");
        ESP.reset();
    }else{
        dbgl("WARNING:OTA UPDATE FAILED");
    }
}else{
    dbgl("INFO:NO OTA UPDATE AVAILABLE");
}

// Wifi config
if(!wc.setConfigFromCloud()){
    dbgl("WARNING: Unable to retrive wifi_config from cloud");
}else{
    dbgl("INFO: wifi_config from cloud sucess");
}

if(!wc.storeConfig()){
    dbgl("WARNING: Unable to store wifi_config to SPIFFS.");
}else{
    dbgl("INFO: wifi_config stored to SPIFFS");
}
// User config
if(!cu.setConfigFromCloud()){
    debugLogF("WARNING: Unable to retrive config_usr from cloud");
}else{
    debugLogF("INFO: Success got config_usr from cloud.");
}

if(!cu.storeConfig()){
    debugLogF("WARNING: Unable to store config_usr to SPIFFS.");
}else{
    debugLogF("INFO: config_usr store to SPIFFS sucess");
}
return true;

wl_status_t wifiSetupSequence(){
    // Initialize WiFi
    WiFi.mode(WIFI_STA);
    // Connect with SPIFFS stored WiFi config

    if(wc.loadConfig()){
        if(!isSSIDAvailable(wc.getSsid())){
            debugLogF("WARNING: SSID not found! Aborting connection try
            with following credentials.");
            debugLogSensitive(String("INFO: SSID:")+wc.getSsid()+"PASS:"+wc.getPasswd());
        }else{
            if(wifiConnect(wc.getSsid(),wc.getPasswd()) != WL_CONNECTED){
                debugLogF("WARNING: Unable to connect with SPIFFS stored
                config.");
            }
        }
    }
}
debugLogSensitive(String("INFO: SSID:")+wc.getSsid()+"PASS:"+wc.getPasswd());
#ifndef PRINT_DEBUG
    WiFi.printDiag(Serial);
#endif
} else{
    debugLogF("INFO: Connected to WiFi!.");
    debugLogSensitive(String("INFO: SSID:")+wc.getSsid()+"PASS:"+wc.getPasswd());
    #ifdef PRINT_DEBUG
        WiFi.printDiag(Serial);
    #endif
    return WiFi.status();
}

// Default config fallback.
if(WiFi.status() != WL_CONNECTED){
    debugLogF("INFO: Falling back to device default WiFi credentials.");

    if(wifiConnectList(defaultWiFiSSID, defaultWiFiPasswd, defaultWiFiListLength) != WL_CONNECTED){
        debugLogF("WARNING: Unable to connect with default configs.");
    } else{
        debugLogF("INFO: Connected to WiFi!.");
        #ifdef PRINT_DEBUG
            WiFi.printDiag(Serial);
        #endif
        return WiFi.status();
    }
}
// Fallback to WiFiManager (works on ESP8266 only)
#if defined(ESP8266)
if(digitalRead(defaultWifiManagerPin) == 0) {
    WiFiManager wifiManager;
    if(WiFi.status() != WL_CONNECTED) {
        debugLogF("INFO:Falling Back to WiFiManager");
        debugLog("INFO: Password:");
        debugLog(defaultWiFiManagerPasswd);
        if(!wifiManager.autoConnect(defaultWiFiManagerSSID, defaultWiFiManagerPasswd)) {
            debugLogF("WARNING: Unable to connect to WiFi through WiFi Manager");
            return WiFi.status();
        } else {
            #ifdef PRINT_DEBUG
            WiFi.printDiag(Serial);
            #endif
        }
    }
}
#endif

return WiFi.status();

/**
 * @brief
 *
 * @param timeout
 * @return wi_status_t
wl_status_t wifiConnectList(const char* ssidList[], const char* passwdList[], int listSize, unsigned long timeout) {

for(int i = 0; i < listSize; i++){
    if(!ssidList[i] || !passwdList[i]){
        continue;
    }

    if(isSSIDAvailable(ssidList[i])){
        if(wifiConnect(ssidList[i], passwdList[i], timeout) == WL_CONNECTED){
            return WL_CONNECTED;
        }
    }
}

return WiFi.status();
}

/**
 * @brief Connect to WiFi
 *
 * @param ssid
 * @param passwd
 * @param timeout WiFi Connection timeout in milli seconds (should be >100)
 * @return WiFi.status() after timeout or connection.
 */

wl_status_t wifiConnect(String ssid, String passwd, unsigned long timeout){

}
return wifiConnect(ssid.c_str(), passwd.c_str(), timeout);
}

wl_status_t wifiConnect(const char* ssid, const char* passwd, unsigned long timeout)
{
    WiFi.begin(ssid, passwd);
    unsigned long t = millis();
    while (WiFi.status() != WL_CONNECTED && millis()-t < timeout)
    {
        delay(100);
    }
    delay(7000);
    return WiFi.status();
}

bool isSSIDAvailable(String ssid){
    for(int n = WiFi.scanNetworks(); n>0; n--){
        if(WiFi.SSID(n-1).equals(ssid))
            return true;
    }
    return false;
}

bool initWifi(){
    if(!WiFi.mode(WIFI_STA)){
        return false;
    }
    WiFi.disconnect();
    return true;
}
bool initSPIFFS(bool format=false){
    if(!SPIFFS.begin())
        return false;
    if(format)
        SPIFFS.format();
    return true;
}

void interruptHandler(){
    ++interruptCounter;
}

config_local.cpp
#include "config_local.h"
#include <ArduinoJson.h>
#include "FS.h"
#if !defined(ESP8266)
    #include "SPIFFS.h"
#endif
#include "utils.h"

configusr::configusr(){
    configId = DEFAULT_CONFIG_ID;
    dataRetrivalTimeMs = DEFAULT_DATA_RETRIVAL_TIME_MS;
    sleepTimeMs = DEFAULT_SLEEP_TIME_MS;
}

configusr::configusr(unsigned long configId, unsigned long dataRetrivalTimeMs, unsigned long sleepTimeMs){
this->configId = configId;
this->dataRetrivalTimeMs = dataRetrivalTimeMs;
this->sleepTimeMs = sleepTimeMs;
}

config_usr::config_usr(const config_usr& c){
    configId = c.configId;
    dataRetrivalTimeMs = c.dataRetrivalTimeMs;
    sleepTimeMs = c.sleepTimeMs;
}

bool config_usr::setConfig(JsonObject& JsonConfig){
    if(JsonConfig["configId"].as<unsigned long>()
        && JsonConfig["dataRetrivalTimeMs"].as<unsigned long>()
        && JsonConfig["sleepTimeMs"].as<unsigned long>()) {
        configId = JsonConfig["configId"].as<unsigned long>();
        dataRetrivalTimeMs = JsonConfig["dataRetrivalTimeMs"].as<unsigned long>();
        sleepTimeMs = JsonConfig["sleepTimeMs"].as<unsigned long>();
        return true;
    }else{
        return false;
    }
}

bool config_usr::setConfig(unsigned long configId, unsigned long dataRetrivalTimeMs, unsigned long sleepTimeMs){
    this->configId = configId;
    this->dataRetrivalTimeMs = dataRetrivalTimeMs;
    this->sleepTimeMs = sleepTimeMs;
    return true;
}
void config usr::getConfig(JsonObject& JsonConfig){
    JsonConfig["configId"] = (configId);
    JsonConfig["dataRetrivalTimeMs"] = (dataRetrivalTimeMs);
    JsonConfig["sleepTimeMs"] = (sleepTimeMs);
}

// TODO: Check if something fails
bool config_usr::storeConfig(){
    StaticJsonBuffer<500> jb;
    JsonObject& obj = jb.createObject();
    this->getConfig(obj);
    String tmp;
    obj.printTo(tmp);
    jb.clear();

    Serial.println("PRININT TO FILE");
    Serial.println(tmp);

    File f = SPIFFS.open(spiffsFileName,"w");
    if(f)
        Serial.println("FILE OPENED");
        f.print(tmp);
        f.close();
        return true;
}

bool config_usr::loadConfig(){
    if(!this->isStored())
    {
        return false;
    }
String tmp;
File f = SPIFFS.open(spiffsFileName,"r");
if(f.available()){
    tmp = f.readStringUntil(\n);  
    Serial.println("READING THIS:::"+tmp);
}else{
    f.close();
    return false;
}
f.close();
    StaticJsonBuffer<500> jb;
JsonObject& obj = jb.parseObject(tmp);
    if(obj.success()){
        this->setConfig(obj);
        jb.clear();
        return true;
    }else{
        jb.clear();
        return false;
    }
}

bool config_usr::isStored(){
    return SPIFFS.exists(spiffsFileName);
}

bool config_usr::setConfigFromCloud(){
    String response;
    StaticJsonBuffer<500> jb;

    int returnCode = httpGET(CONFIG_USR_ENDPOINT, response);
if(returnCode == 200){
    JsonObject& obj = jb.parseObject(response);
    if(!obj.success()){
        jb.clear();
        Serial.println("config_usr::setConfigFromCloud JSON parsing failed!");
        return false;
    }
    if(obj["response_code"].as<int>() == CONFIG_RESPONSE_OK){
        bool s = this->setConfig(obj);
        if (!s)
        {
            Serial.println("config_usr::setConfigFromCloud Object structure invalid");
            obj.prettyPrintTo(Serial);
            jb.clear();
            return false;
        }
        jb.clear();
        return true;
    }
    if(obj["response_code"].as<int>() == NO_CONFIG_UPDATE){
        Serial.println(F("config_usr::setConfigFromCloud no config update"));
        obj.prettyPrintTo(Serial);
        jb.clear();
        return true;
    }
    Serial.println("config_usr::setConfigFromCloud Config response not OK");
    obj.prettyPrintTo(Serial);
    jb.clear();}
return false;
}
Serial.println("config_usr::setConfigFromCloud returncode "+String(returnCode));
jb.clear();
return false;
}
wifi_config::wifi_config(){
    ssid = String(DEFAULT_SSID);
    passwd = String(DEFAULT_PASSWD);
}

wifi_config::wifi_config(const char* s, const char* p){
if(s && p){
    ssid = String(s);
    passwd = String(p);
}else{
    ssid = String(DEFAULT_SSID);
    passwd = String(DEFAULT_PASSWD);
}
}

wifi_config::wifi_config(JsonObject& JsonWifiConfig){
if(JsonWifiConfig["ssid"].as<char*>() && JsonWifiConfig["passwd"].as<char*>()){
    ssid = String(JsonWifiConfig["ssid"].as<char*>());
    passwd = String(JsonWifiConfig["passwd"].as<char*>());
}else{
    ssid = String(DEFAULT_SSID);
    passwd = String(DEFAULT_PASSWD);
}
bool wifi_config::setConfig(const char* s, const char* p){
    if (!s || !p)
        return false;

    ssid = String(s);
    passwd = String(p);
    return true;
}

bool wifi_config::setConfig(JsonObject& JsonWifiConfig){
    if(JsonWifiConfig["ssid"].as<char*>() &&
    JsonWifiConfig["passwd"].as<char*>()){
        ssid = String(JsonWifiConfig["ssid"].as<char*>());
        passwd = String(JsonWifiConfig["passwd"].as<char*>());
        return true;
    }else{
        return false;
    }
}

bool wifi_config::getConfig(JsonObject& JsonWifiConfig){
    JsonWifiConfig["ssid"] = ssid;
    JsonWifiConfig["passwd"] = passwd;
    return true;
}

String wifi_config::getPasswd(){
    return passwd;
}

String wifi_config::getSsid(){
return ssid;
}

wifi_config::~wifi_config(){
}

bool wifi_config::storeConfig(){
    StaticJsonBuffer<100> jb;
    JsonObject& obj = jb.createObject();
    this->getConfig(obj);
    String tmp;
    obj.printTo(tmp);
    jb.clear();

    File f = SPIFFS.open(spiffsFileName,"w");
    if(!f)
        return false;
    f.print(tmp);
    f.close();
    return true;
}

bool wifi_config::loadConfig(){
    if(!this->isStored())
    {
        return false;
    }

    File f = SPIFFS.open(spiffsFileName,"r");
    String tmp = f.readStringUntil('n');
    f.close();
StaticJsonBuffer<100> jb;
JsonObject& obj = jb.parseObject(tmp);

if(obj.success()){
    this->setConfig(obj);
    jb.clear();
    return true;
} else{
    jb.clear();
    return false;
}

bool wifi_config::isStored(){
    return SPIFFS.exists(spiffsFileName);
}

bool wifi_config::setConfigFromCloud(){
    String response;
    StaticJsonBuffer<500> jb;

    int returnCode = httpGET(WIFI_CONFIG_ENDPOINT, response);

    if(returnCode == 200){
        JsonObject& obj = jb.parseObject(response);

        if(!obj.success()){
            jb.clear();
            Serial.println("wifi_config::setConfigFromCloud JSON parsing failed!");
            return false;
        }
    }
if(obj["response_code"].as<int>() == CONFIG_RESPONSE_OK){
    bool s = this->setConfig(obj);
    if (!s)
    {
        Serial.println("wifi_config::setConfigFromCloud Object
structure invalid");
        obj.prettyPrintTo(Serial);
        jb.clear();
        return false;
    }
    jb.clear();
    return true;
}

if(obj["response_code"].as<int>() == NO_CONFIG_UPDATE){
    Serial.println(F("wifi_config::setConfigFromCloud no config
update"));
    obj.prettyPrintTo(Serial);
    jb.clear();
    return true;
}

    Serial.println("wifi_config::setConfigFromCloud Config response
not OK");
    obj.prettyPrintTo(Serial);
    jb.clear();
    return false;
}
    Serial.println("wifi_config::setConfigFromCloud returncode
"+String(returnCode));
    jb.clear();
    return false;
config_local.h

#ifndef _CONFIG_LOCAL_H_
#define _CONFIG_LOCAL_H_

#include <ArduinoJson.h>

#define DEFAULT_DATA_RETRIVAL_TIME_MS 6000
#define DEFAULT_SLEEP_TIME_MS 1200
#define DEFAULT_CONFIG_ID 0
#define DEFAULT_SSID "test"
#define DEFAULT_PASSWD "test1234567890"

#ifdef USE_HTTP_BIN
#define WIFI_CONFIG_ENDPOINT "http://httpbin.org/anything"
#define CONFIG_USR_ENDPOINT "http://httpbin.org/anything"
#else
#define WIFI_CONFIG_ENDPOINT "http://<Server>/wifi_config"
#define CONFIG_USR_ENDPOINT "http://<Server>/config_usr"
#endif

#define CONFIG_RESPONSE_OK 1
#define NO_CONFIG_UPDATE 2

class config_usr
{
    unsigned long configId;
    unsigned long dataRetrivalTimeMs;
    unsigned long sleepTimeMs;
    String spiffsFileName = "/config_usr.json";

public:
    config_usr();
    config_usr(unsigned long configId, unsigned long dataRetrivalTimeMs, unsigned long sleepTimeMs);
config_usr(const config_usr& c);
// config(JsonObject& JsonConfig);

bool setConfig(unsigned long configId, unsigned long
dataRetrivalTimeMs, unsigned long sleepTimeMs);
bool setConfig(JsonObject& JsonConfig);
void getConfig(JsonObject& JsonConfig);
bool setConfigFromCloud();

// Load and store config to SPIFFS
bool storeConfig();
bool loadConfig();
bool isStored();

String getSpiffsFileName(){
    return String(spiffsFileName);
}

void setSpiffsFileName(String f){
    spiffsFileName = f;
}

unsigned long getConfigId(){
    return configId;
}

unsigned long getDataRetrivalTimeMs(){
    return dataRetrivalTimeMs;
}

unsigned long getSleepTimeMs(){
    return sleepTimeMs;
}
void setConfigId(unsigned long c) {
    configId = c;
}

void setDataRetrivalTimeMs(unsigned long d) {
    dataRetrivalTimeMs = d;
}

void setSleepTimeMs(unsigned long s) {
    sleepTimeMs = s;
}

class wifi_config {
    String ssid;
    // TODO: Secure password storing method.
    String passwd;
    bool isSet;
    String spiffsFileName = "/wifi_config.json";

public:
    wifi_config();
    wifi_config(const char* s, const char* p);
    wifi_config(JsonObject& JsonWifiConfig);

    bool setConfig(const char* s, const char* p);
    bool setConfig(JsonObject& JsonWifiConfig);
    bool getConfig(JsonObject& JsonWifiConfig);
    bool setConfigFromCloud();

    // Load and store config to SPIFFS
    bool storeConfig();
    bool loadConfig();
bool isStored();

String getSpiffsFileName(){
    return String(spiffsFileName);
}

void setSpiffsFileName(String f){
    spiffsFileName = f;
}
String getSsid();
String getPasswd();
~wifi_config();

#endif

data_payload.cpp

#include "data_payload.h"
#include <cstdlib>
#include "utils.h"
#include "FS.h"

#if !defined(ESP8266)
    #include "SPIFFS.h"
#endif

data_payload::data_payload() {
    data = 0;
    tdiff = 0;
    t = 0;
    isDataSet = false;
}
data_payload::data_payload(unsigned long d, unsigned long td,
unsigned long tm) {
    data = d;
    tdiff = td;
    t = tm;
    isDataSet = true;
}

bool data_payload::append_data(unsigned long d, unsigned long td,
unsigned long tm) {
    data = d;
    tdiff = td;
    t = tm;
    isDataSet = true;
    return true;
}

bool data_payload::isStored() {
    return SPIFFS.exists(spiffsFileName);
}

bool data_payload::storeSpiffs() {
    if (!isDataSet)
        return false;
    File f = SPIFFS.open(spiffsFileName, "a");
    if (!f)
        return false;
    f.print(String(data) + "," + String(tdiff) + "," + String(t));
    f.print(\n);
    f.close();
return true;
}

// Just loads the last Data!
// Unused
bool data_payload::loadSpiffs() {
  if (!isStored())
    return false;

  File f = SPIFFS.open(spiffsFileName, "r");
  if (!f)
    return false;

  String line, d, tm, td;

  while (f.available()) {
    line = f.readStringUntil('n');
    d = getValue(line, ',', 0);
    td = getValue(line, ',', 1);
    tm = getValue(line, ',', 2);
    if ((d == "") || (tm == "") || (td == "")) {
      f.close();
      return false;
    }
    if (!append_data(strtoul(d.c_str(), NULL, 0),
                     strtoul(td.c_str(), NULL, 0), strtoul(tm.c_str(), NULL, 0))) {
      f.close();
      return false;
    }
  }
  f.close();
  return true;
bool data_payload::sendToCloud() {

    String s;
    _to_json_hack(s);
    if (s == "")
        return false;

    String r;

    int returnCode = httpPOST(DATA_PAYLOAD_ENDPOINT, s, r);
    StaticJsonBuffer<500> jb;
    if (returnCode == 200) {
        JsonObject& obj = jb.parseObject(r);
        if (!obj.success()) {
            jb.clear();
            Serial.println("Couldn't parse json");
            return false;
        }

        if (obj["response_code"].as<int>() == DATA_RESPONSE_OK) {
            jb.clear();
            return true;
        }
    }
    jb.clear();
    return false;
}

// it actually makes a csv
void data_payload::_to_json_hack(String& s) {
    s = "";
    if(!isStored()){
        return;
    }

    File f = SPIFFS.open(spiffsFileName,"r");
    if(!f){
        return;
    }

    String line, d, tm, td;
    s = "data,tdiff,timestamp\n";
    while(f.available()){
        line = f.readStringUntil(\'\n\');
        d = getValue(line, ',', 0);
        td = getValue(line, ',', 1);
        tm = getValue(line, ',', 2);
        if ((d == "") || (tm == "") | (td == ") {  
                f.close();
                s="";
                rmSpiffs();
                return;
            }
        s += String(d)+","+String(td)+","+String(tm)+"\n";
    }
    return;
}

void data_payload::flush() {
    data = 0;
tdiff = 0;
t = 0;
isDataSet=false;
}

bool data_payload::rmSpiffs() {
    if (!isStored())
        return true;
    return SPIFFS.remove(spiffsFileName);
}

data_payload.h

#define DATA_PAYLOAD_H_

#ifdef USE_HTTP_BIN
    #define DATA_PAYLOAD_ENDPOINT "http://httpbin.org/anything"
#else
    #define DATA_PAYLOAD_ENDPOINT "http://<Server>/data_payload"
#endif

#define DATA_RESPONSE_OK 1
#include <Arduino.h>
#include <ArduinoJson.h>

class data_payload{
    unsigned long data;
    unsigned long tdiff;
    unsigned long t;
    String spiffsFileName = "/data_payload.json";

    bool isDataSet = false;
public:
    data_payload();
    data_payload(unsigned long d, unsigned long td, unsigned long tm);
    bool append_data(unsigned long d, unsigned long td, unsigned long tm);

    bool getJson(JsonObject& JsonData, unsigned int index);

    bool sendToCloud();
    void _to_json_hack(String& s);

    bool storeSpiffs();
    bool loadSpiffs();
    bool isStored();

    String getSpiffsFileName(){
        return spiffsFileName;
    }

    void setSpiffsFileName(String f){
        spiffsFileName = f;
    }

    unsigned long getData(){
        return data;
    }

    unsigned long getTimediff(){
        return tdiff;
    }

    unsigned long getTimestamp(){

void flush();
bool rmSpiffs();

#endif

ota.cpp
#include "ota.h"
#include "utils.h"
#include <ESP8266httpUpdate.h>
#include <ESP8266HTTPClient.h>

bool checkForUpdate(){
  String response;
  int response_code = httpGET(OTA_CHECK_ENDPOINT, response);

  Serial.println(String("OTA UPDATE RESPONSE:" ) + response);

  if(response_code != 200){
    Serial.println("Response code invalid");
    return false;
  }

  if(response.toInt() > currentVersion){
    Serial.println("CURRENT VERSION:"+String(currentVersion));
    Serial.println("ONLINE VERSION:"+String(response.toInt()));
    return true;
  }
  return false;
}
bool doUpdateOTA(){
    Serial.println(OTA_BIN_ENDPOINT);
    t_httpUpdate_return ret = ESPhttpUpdate.update(OTA_BIN_ENDPOINT);
    Serial.println("UPDATE CODE" + String(ret));
    switch(ret){
        case HTTP_UPDATE_FAILED:
            Serial.println("Update failed!");
            Serial.println(ESPhttpUpdate.getLastError());
            Serial.println(ESPhttpUpdate.getLastErrorString());
            return false;
            break;
        case HTTP_UPDATE_NO_UPDATES:
            Serial.println("NO updates");
            return false;
            break;
        case HTTP_UPDATE_OK:
            Serial.println("UPDATE SUCESS");
            return true;
            break;
    }
    return false;
}

ota.h

#ifndef _OTA_H
#define _OTA_H

#define OTA_CHECK_ENDPOINT "http://<Server>/ota_check"
#define OTA_BIN_ENDPOINT String("http://<Server>/")+
DEFAULT_HTTP_TOKEN +"/ota.bin"
#include <Arduino.h>
extern const int currentVersion;
bool checkForUpdate();
bool doUpdateOTA();
#endif

utils.cpp

#include "utils.h"
#include "FS.h"
#if defined(ESP8266)
  #include "ESP8266HTTPClient.h"
#else
  #include "HTTPRequest.h"
  #include "SPIFFS.h"
#endif
#include <ArduinoJson.h>

String getValue(String data, char separator, int index)
{
  int found = 0;
  int strIndex[] = {0, -1};
  int maxIndex = data.length()-1;

  for(int i=0; i<=maxIndex && found<=index; i++){
    if(data.charAt(i)==separator || i==maxIndex){
      found++;
      strIndex[0] = strIndex[1]+1;
      strIndex[1] = (i == maxIndex) ? i+1 : i;
    }
  }

  return found>index ? data.substring(strIndex[0], strIndex[1]) : "";
}
int storeJson(const char* fileName, const JsonObject& obj) {
    String tmp;
    obj.printTo(tmp);
    return writeFile(fileName,tmp);
}

String readFile_usr(const char* fileName) {
    String buf;
    if(!SPIFFS.exists(fileName))
        return String("'");
    File f = SPIFFS.open(fileName,"r");
    if(!f)
        return String("'");
    if(f.available())
        buf = f.read();
    else
        return String("'");
    return buf;
}

int writeFile(const char* fileName, String buf) {
    File f = SPIFFS.open(fileName,"w");
    if(!f)
        return FILE_CANT_OPEN;
    f.print(buf);
    f.close();
    return SUCESS;
}

int httpPOST(const char* url, String payload, String &response) {
    HTTPClient http;
    if(!http.begin(url)) {
        return 0;
    }
http.addHeader("Token",DEFAULT_HTTP_TOKEN);
int returnCode = http.POST(payload);
response = http.getString();
Serial.println("\n\n\n:::::::::::::::HTTP POST
PAYLOAD:::::::"+String(payload));
Serial.println("\n\n\n:::::::::::::::HTTP POST
URL:::::::"+String(url));
Serial.println("\n\n\n:::::::::::::::HTTP POST
RESPONSE:::::::"+response);
http.end();
return returnCode;
}

int httpPOSTJson(const char* url, String payload, String &response){
    HTTPClient http;
    if(!http.begin(url)){
        return 0;
    }
    http.addHeader("Content-Type","application/json");
    http.addHeader("Token",DEFAULT_HTTP_TOKEN);
    int returnCode = http.POST(payload);
    response = http.getString();
    Serial.println("\n\n\n:::::::::::::::HTTP POST
PAYLOAD:::::::"+String(payload));
    Serial.println("\n\n\n:::::::::::::::HTTP POST
URL:::::::"+String(url));
    Serial.println("\n\n\n:::::::::::::::HTTP POST
RESPONSE:::::::"+response);
    http.end();
    return returnCode;
}
int httpPOSTJson(const char* url, JsonObject& obj, String &response){
    String payload;
    obj.printTo(payload);
    return httpPOSTJson(url, payload, response);
}

int httpGET(const char* url, String& response){
    HTTPClient http;
    if(!http.begin(url)){
        return 0;
    }
    http.addHeader("Token",DEFAULT_HTTP_TOKEN);
    int returnCode = http.GET();
    response = http.getString();
    Serial.println("\n\n\n:::::HTTP GET URL:::"+String(url));
    Serial.println("\n\n\n:::::HTTP GET RESPONSE:::"+response);
    http.end();
    return returnCode;
}

int httpGETJson(const char* url, JsonObject& obj, String &response){
    String dataLink = String(url)+"?";
    JsonObject::iterator it;
    for (it = obj.begin(); it!= obj.end(); ++it){
        dataLink += it->key + String("=") + it->value.as<char*>() + ";
    }
}
```cpp
int storeJson(const char* fileName, const JsonObject& obj);
#define loadJson(fileName,jb)
jb.parseObject(readFile_usr(fileName))
String readFile_usr(const char* fileName);
int writeFile(const char* fileName, String buf);
int httpPOST(const char* url, String payload, String &response);
#define httpPOSTJson(url, payload, &response)
int httpPOSTJson(const char* url, String payload, String &response);
int httpGET(const char* url, String &response);
```
int httpGETJson(const char* url, JsonObject& obj, String &response);
String getValue(String data, char separator, int index);
#endif
index.js

const express = require('express');
const MongoClient = require('mongodb').MongoClient;
const csv = require('csvtojson');
const bodyParser = require('body-parser');
const crypto = require('crypto');
const fs = require('fs');
const multer = require('multer');

const upload = multer({
    dest: './ota_bin'
});

const mongouri2 = process.env.MONGO_URI;

const app = express();
app.listen(process.env.PORT || 5000, process.env.HOST||'0.0.0.0', ()
=> {
    console.log('listening on someport'+process.env.PORT || 5000)
});

MongoClient.connect(mongouri2, (err, client) => {
    if (err) return console.log(err)
db = client.db('main');
}

app.use(bodyParser.json()); // for parsing application/json
//app.use(rawBody);

app.set('view engine', 'ejs');

app.get('/',async (req,res) =>{
    res.sendFile(__dirname + '/index.html');
});

app.get('/get_config/:token', async (req,res) =>{
    var config_usr
    try{
        config_usr= await db.collection("config_usr").findOne({"token":req.params.token});
    }catch (e){
        console.log(e);
        config_usr = {configId:0,dataRetrivalTimeMs:0,sleepTimeMs:0,token:0};
    }

    var wifi_config
    try{
        wifi_config = await db.collection("wifi_config").findOne({"token":req.params.token});
        wifi_config.passwd = crypto.createHash('sha256').update(wifi_config.passwd).digest('hex');
    }catch (e){
        console.log(e);
    }
})
```javascript
wifi_config = {ssid:"n",passwd:"n",token:'n'};
}

var data_payload;
try{
    data_payload = await 
    db.collection('data_payload').find({"token":req.params.token}).toAr 
    ray();
}catch (e){
    console.log(e);
    data_payload = [{data:0,tdiff:0,timestamp:0,token:0}];
}

res.render('configPage.ejs',{config_usr:config_usr,
    wifi_config:wifi_config,data_payload:data_payload,token:req.params.
    token});
}

app.get('/set_config/config_usr/', async (req,res)=>{
    var config_usr;
    try{
        config_usr = await 
        db.collection('config_usr').findOne({token:req.query.token});
    }catch (e){
        console.log(e);
        res.status(501).end();
    }

    if(!config_usr){
        try{
            const dbres = await db.collection("config_usr").save( 
            {
```
token: req.query.token,
    configId: req.query.configId,
    dataRetrivalTimeMs: req.query.dataRetrivalTimeMs,
    sleepTimeMs: req.query.sleepTimeMs
});

res.send({response_code:1}).end();

}catch(e){
    console.log(e);
    res.status(502).end();
}
}

} else {
    try{
        const dbres = await db.collection("config_usr").updateOne({token:req.query.token},
            {$set: {
                token: req.query.token,
                configId: req.query.configId,
                dataRetrivalTimeMs: req.query.dataRetrivalTimeMs,
                sleepTimeMs: req.query.sleepTimeMs
            }});
    } catch (e){
        // res.send(e);
        res.status(500).end();
    }
}

res.json({response_code:1});
}
app.get('/set_config/wifi_config/', async (req,res) => {
    var wifi_config;
    try{
        wifi_config = await db.collection('wifi_config').findOne({token:req.query.token});
    } catch (e){
        console.log(e);
        res.status(501).end();
    }

    if(!wifi_config){
        try{
            const dbres = await db.collection("wifi_config").save(
                {
                    token: req.query.token,
                    ssid: req.query.ssid,
                    passwd: req.query.passwd,
                });
            res.send({response_code:1}).end();
        } catch(e){
            console.log(e);
            res.status(502).end();
        }
    }

    if(wifi_config){
        try{
            const dbres = await db.collection("wifi_config").updateOne({token:req.query.token),
        } catch(e){
            console.log(e);
            res.status(502).end();
        }
    }
})}
app.get('/config_usr', async (req, res) => {
  const receivedToken = req.get('Token');
  db.collection('config_usr').findOne({token: receivedToken},
    async (err, result) => {
      if (err) {
        res.json({response_code: 2});
        console.error(err);
      } else {
        console.log('Got Result: ')
        console.log(result);
        res.json({response_code: 1,
          configId: result.configId,}
dataRetrivalTimeMs: result.dataRetrivalTimeMs,
sleepTimeMs: result.sleepTimeMs
});

app.get('/wifi_config', async (req, res) => {
    const receivedToken = req.get('Token');
    db.collection('wifi_config').findOne({token: receivedToken}, async (err, result) => {
        if (err) {
            res.json({response_code: 2});
            console.error(err);
        } else {
            console.log('Got Result: ');
            console.log(result);
            res.json({
                response_code: 1,
                ssid: result.ssid,
                passwd: result.passwd
            });
        }
    });
});
function rawBody(req, res, next) {
  req.setEncoding('utf8');
  req.rawBody = '';
  req.on('data', function(chunk) {
    req.rawBody += chunk;
  });
  req.on('end', function() {
    next();
  });
}

app.post('/data_payload', rawBody, async (req, res) => {
  const receivedToken = req.get('Token');
  if (!receivedToken) {
    console.log('Response code2');
    res.json({response_code: 2});
  }
  console.log(req.rawBody);
  var jsonArray;
  try {
    jsonArray = await csv().fromString(req.rawBody);
    console.log(jsonArray);
  } catch (e) {
    console.log(e);
    res.json({response_code: 2});
    return;
  }

  jsonArray.forEach(async (el) => {
    // Further processing
  });
db.collection('data_payload').save(
    {token: receivedToken,
    data: el.data,
    tdiff: el.tdiff,
    timestamp: el.timestamp
    }, async (error, result) => {
    if (error)
        console.log(error);
    //console.log(result);
    console.log("Yo");
    })
    })
    } res.json({response_code:1});
})

app.post('/ota_upload/:token', upload.any(), async (req, res, next) => {
    const receivedToken = req.params.token;
    const fp = req.files[0];
    const fileName = fp.filename;
    const olddir = fp.destination;
    try {
        fs.mkdir(olddir + '/' + receivedToken,{recursive:true},
        (err) => {
            if (err){
                console.error(err);
                res.sendStatus(501).end();
            }
        })
    }
fs.rename(olddir + "/' + fileName, olddir + '" + receivedToken + "'/ota.bin', (error)=>{
  if (error){
    console.log(error);
    res.sendStatus(501).end();
  }
  res.send("Done Uploading");
});

} catch (e) {
  console.log(e);
  res.sendStatus(501).end();
}

console.log(fp);

app.get('/ota_check', async (req,res)=>{
  var receivedToken = req.get('Token');
  const path = './ota_bin/'+receivedToken+'/ota.bin';
  if(req.query.version === undefined){
    // If there is request from device
    try{
      db.collection('ota').findOne({token:receivedToken},
        async (error,result)=>{
          if(error){
            console.error(err);
            res.sendStatus(404).end();
          }
        });
    }
  }
});
80

```javascript
} else {
    console.log('Got Result:');
    console.log(result);
    res.send(result.version);
}
```

```javascript
})
}
```
app.get('/:token/ota.bin', async (req, res) => {
    const receivedToken = req.params.token;
    const path = './ota_bin/' + receivedToken + '/ota.bin';
    if (fs.existsSync(path)) {
        res.sendFile(path, {
            root: __dirname
        });
    } else {
        try {
            const dbres = await db.collection("ota").updateOne({token: receivedToken},
                {
                    $set: {
                        version: req.query.version
                    }
                });
            res.send({response_code: 1}).end();
        } catch (e) {
            // res.send(e);
            res.status(500).end();
        }
    }
});
}, (err) => {
    if (err) {
        console.log(err);
    } else {
        console.log('File sent!: ' + path);
    }
});
}
}
else {
    res.sendStatus(304).end();
}
});

**configPage.ejs**

<h2>Basic Config for Device Id: <%= token %></h2>

<h2>Basic Config</h2>

<form class="set_config_usr" target="_blank" action="/set_config/config_usr" method="GET">
  <input name="token" placeholder="device id" required="required"/>
  <input name="configId" placeholder="configId" required="required" pattern="[0-9]{1,}"
  
  <input name="dataRetrivalTimeMs" placeholder="data retrival time in ms" required="required" pattern="[0-9]{1,}"
  
  <input name="sleepTimeMs" placeholder="sleep time in ms" required="required" pattern="[0-9]{1,}"
  
  <button id="submit_config_usr">Submit</button>
</form>
<h2>WiFi Config</h2>
<form class="set_wifi_config" target="_blank" action="/set_config/wifi_config">
  <input name="token" placeholder="device id" required="required"/>
  <input name="ssid" placeholder="ssid">
  <input name="passwd" placeholder="passwd">
  <button id="submit_wifi_config">Submit</button>
</form>

<h2>OTA Update</h2>
<form class="ota_update_upload" target="_blank" action="/ota_upload/config_usr.token" enctype="multipart/form-data" method="POST">
  Firmware Bin File:<input type="file" name="otaBinFile">
  <input type="submit">
</form>

<h2>Set OTA Version</h2>
<form class="set_ota_version" target="_blank" action="/ota_check" method="GET">
  Version:<input name="version" type="number" placeholder="version number">
  Token:<input name="token" type="text" placeholder="token">
  <input type="submit">
</form>

<table class="config_usr">
  <tr>
    <th>configId</th>
  </tr>
  <tr>
    <td>configId</td>
  </tr>
</table>
<table>
<thead>
<tr>
<th>dataRetrivalTimeMs</th>
<th>sleepTimeMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%= config_usr.configId %&gt;</td>
<td>&lt;%= config_usr.dataRetrivalTimeMs %&gt;</td>
</tr>
<tr>
<td>&lt;%= config_usr.sleepTimeMs %&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ssid</th>
<th>password (sha256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%= wifi_config.ssid %&gt;</td>
<td>&lt;%= wifi_config.passwd %&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>timestamp</th>
<th>count</th>
<th>counting time</th>
<th>CPM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<% for(var i=data_payload.length -1; i >=0 ;i--) {%>
<tr>
<td><%= data_payload[i].timestamp %></td>
<td><%= data_payload[i].data %></td>
<td><%= data_payload[i].tdiff %></td>
<td><%= 60000*data_payload[i].data / data_payload[i].tdiff %></td>
</tr>
<% } %>
</table>