

# EECE 500 – Approved Experience

## Final Report

*Author:* Hadi Rayan Al-Sandid

*Supervisor:* Dr. Mazen Saghir

*Internship Location:* American University of Beirut

*Training Term:* Summer 2020

*Report Date:* September 8th, 2020

# Table of Contents

## Contents

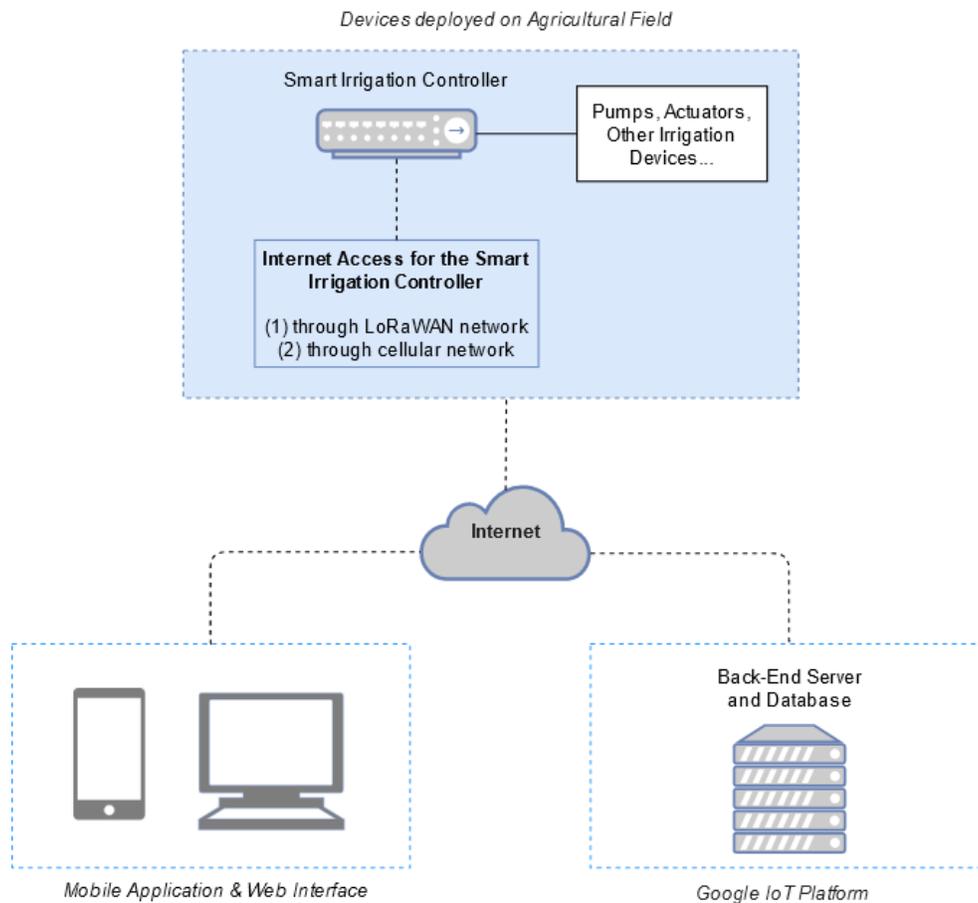
<b>Table of Contents</b> .....	2
<b>Table of Figures</b> .....	3
<b>Introduction</b> .....	4
<b>Training Overview</b> .....	5
Project Overview.....	5
Smart Irrigation Controller.....	5
Selecting base hardware.....	6
Software & Code .....	6
Alternative Power System.....	7
Custom PCB.....	9
Back-End Server .....	12
Web Interface & Mobile Application .....	13
Documentation .....	15
<b>Global, Economic, Environmental, and Societal Impact of Completed Work</b> .....	17
<b>Conclusion</b> .....	18
<b>References</b> .....	20
<b>Appendices</b> .....	21
Appendix A: Background Information on the Project.....	21
Appendix B: Raspberry-Pi Power Requirements Overview .....	22
Appendix C: Raspberry-Pi 3 Model B Overview .....	23
Appendix D: Python3 'Requests' Library Overview .....	24
Appendix E: Python3 'RPI.GPIO' Library Overview .....	25
Appendix F: Back-End API Calls.....	27
Appendix G: Raspberry-Pi – Power Consumption Benchmarks.....	37

## Table of Figures

Figure 1 : Diagram of the Smart Irrigation System .....	4
Figure 2- Diagram of the Smart Irrigation Controller .....	6
Figure 3- Diagram of the Portable Power System .....	8
Figure 4 - MTXDOT-EU1-A00-1 (Left) and NL-SW-GPRS (Right) .....	10
Figure 5- Schematic of our add-on PCB .....	11
Figure 6- 3D Visualization of our add-on PCB.....	12
Figure 7- Web Interface, 'Analytics' page .....	14
Figure 8- Mobile Application.....	15
Figure 9 - Documentation Website.....	16
Figure 10 - Micro-USB port of the Raspberry-Pi .....	22
Figure 11- GPIO Pins used to Power the Raspberry-Pi .....	22
Figure 12- Raspberry-Pi 3 Model B .....	23
Figure 13- Power requirements for different Raspberry-Pi models.....	38

## Introduction

During my internship, I have worked under the supervision of Dr. Mazen Saghir on a Smart Irrigation System. Its main objective is to 'enhance' the irrigation process on agricultural fields, which is carried out by scheduling optimal irrigation cycles to reduce water wastage and maximize crop yield. There are three main components in this system: (1) A Smart Irrigation Controller, (2) a Back-End Server, (3) and a Web Interface & Mobile Application. My work mainly consisted of R&D on all these components, with a pronounced focus on the Smart Irrigation Controller.



*Figure 1 : Diagram of the Smart Irrigation System*

## Training Overview

### Project Overview

My internship took place between the 8<sup>th</sup> June and 8<sup>th</sup> August (i.e. around 8 weeks). For the duration of my internship, I held weekly meeting with my supervisor Dr. Mazen Saghir. These weekly meetings were scheduled on Wednesdays at 11 AM, and have been used to discuss recent progress, review documentation, and talk about the next steps we should take in the development of the Smart Irrigation System.

As mentioned in the introduction, I have worked on the three main components of the system (i.e. Smart Irrigation Controller, Back-End Server, and Web Interface & Mobile Application). We will discuss all these mentioned components shortly.

### Smart Irrigation Controller

The term “Smart Irrigation Controller” represents the combination of (1) a Raspberry-Pi board, (2) all software and code used to run the logic of the controller and its components, (3) a self-sufficient and portable power solution to power the Raspberry-Pi, (4) and an addon PCB for the Raspberry-Pi.

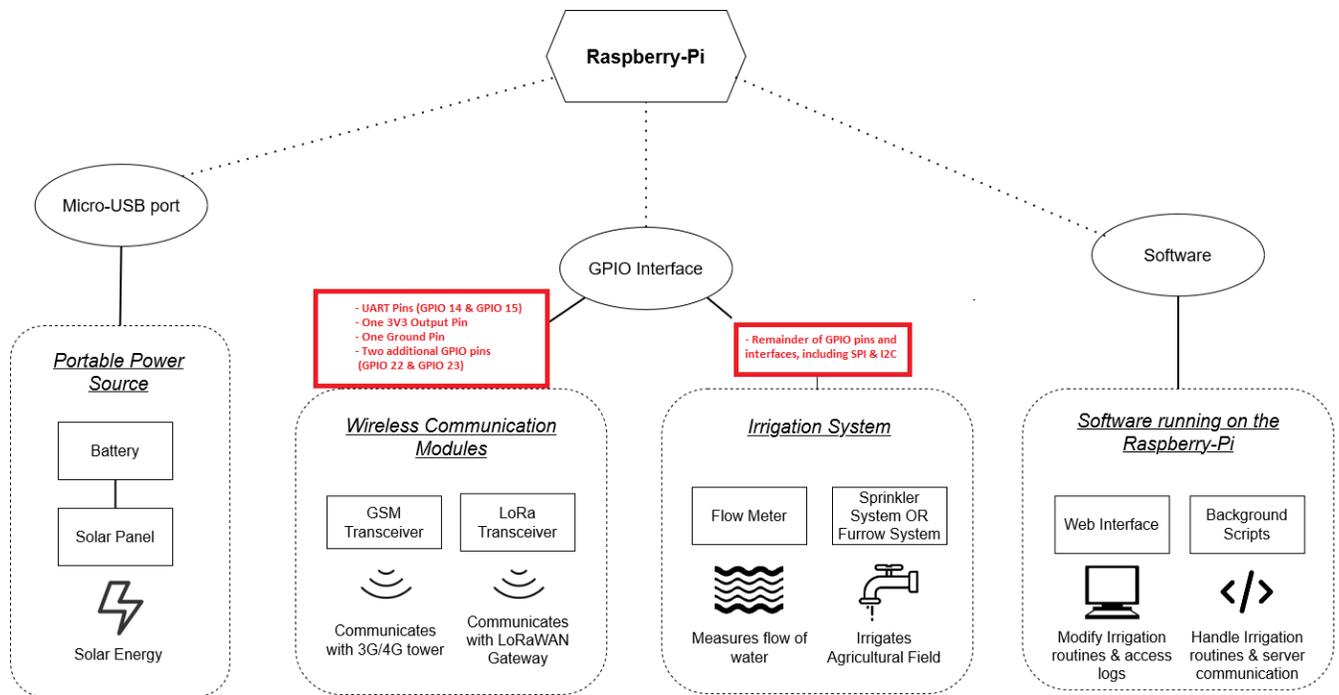


Figure 2- Diagram of the Smart Irrigation Controller

## Selecting base hardware

When selecting the hardware to base our Smart Irrigation Controller on, we wanted a solution which would allow us to run and debug scripts easily, have Wi-Fi connectivity out-of-the-box, offer a large GPIO interface (i.e. more than 20 controllable pins), and be relatively inexpensive (i.e. less than 100 USD). The Raspberry-Pi fit all the criteria with its hardware specifications, so we have decided to use it as the base of our Smart Irrigation Controller. The model we have selected specifically is the *Raspberry-Pi 3 Model B*, on which more details are available in *APPENDIX C*.

## Software & Code

Our Raspberry-Pi model supports most mainstream Linux distributions, which has allowed us to use tools available on these distributions to develop the scripts and logic

associated with our Smart Irrigation Controller in a more straightforward manner. We have decided to go with Raspbian Lite, also called Raspberry-Pi OS, as it is the most optimized Linux distribution for the Pi, and it only contains necessary daemons. During testing, we had no issues running scripts in C, C++, and Python using native support and some additional packages. More specifically, we've used the Python3 Library '*RPI.GPIO*' to write scripts controlling the GPIO interface of the Raspberry-Pi, which is required whenever we'd like the Pi to communicate with external modules or interfaces (see *APPENDIX E* for more an overview of the library). Also, we've used the Python3 Library '*Py.Requests*' to send HTTP requests directly from the Raspberry-Pi to other devices over the internet (see *APPENDIX D* for more an overview of the library). Other software we used include the *Apache* web server program, to host the Web Interface directly on the Raspberry-Pi, and *Incron*, which is a similar program to CRON but instead of running commands based on time, it triggers commands based on file/directory events. In the later stages of the Smart Irrigation System project, we plan to distribute Smart Irrigation Controllers which come pre-packaged with an SD-card containing the OS and all necessary programs, effectively allowing it to run anywhere without any additional setup required from users. All the programs we'd like to package are available under public license and can be used & re-distributed as long as their respective license is included (*Apache* is available under the Apache License 2.0, *Incron* and *Raspbian* are available under the GNU General Public License, version 2).

## Alternative Power System

We plan on deploying the Smart Irrigation Controller directly on agricultural fields. In some cases, this might leave some units far from any power source (i.e. electrical grid).

To power these units, we had to research possible portable and self-sufficient power solutions. One of the power solution we considered is a solar-powered system, featuring a compact solar panel, a battery, and additional circuitry to deliver the right amount of voltage/current to the base of the Smart Irrigation Controller (i.e. the Raspberry-Pi board). Before moving forward with the power system's design, we had to assess the expected power usage of the Raspberry-Pi model we were using. We could not run proper benchmarks in the MSFEA labs due to COVID restrictions, so we had to use benchmark runs from an online source (Source : [PidRamble](#)). The Raspberry-Pi model we are using (i.e. *Raspberry-Pi 3 Model B*) consumes 1.4 Watts when idle (5v@260mA), 1.2W at minimum CPU load (5v@230mA), and 3.7W at maximum CPU load (5v@730mA). See *APPENDIX B* for more details on how the Raspberry-Pi models can be powered (i.e. regular adapter, GPIO pins...) and *APPENDIX G* for more details on the power consumption of different Raspberry-Pi models. While we have conducted research on the solar-powered system, we did not have the opportunity to assemble it and conduct tests due to limited lab access. As such, we have decided to not include the solar-powered system in our first prototype of the Smart Irrigation System, and just use a regular power adapter.

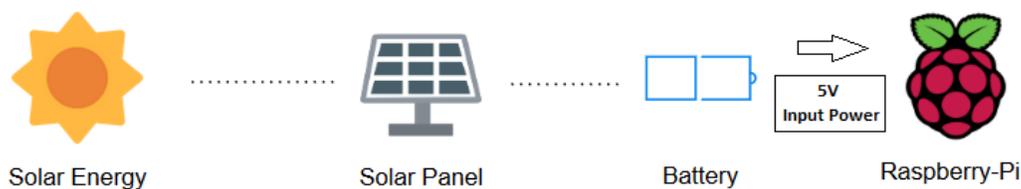


Figure 3- Diagram of the Portable Power System

## Custom PCB

We would like the Smart Irrigation Controller to have constant internet access to be able to communicate with the Back-End Server without any downtime. However, as we have mentioned in the previous section, our units will be deployed directly on agricultural fields, and might be located far from any wireless infrastructure (i.e. more than 1KM). Wireless modules which are built-in the Raspberry-Pi, like Wi-Fi or Bluetooth, are not useful in this case due to their limited range, meaning we would have to look for alternative ways to connect our Smart Irrigation Controller to the internet. LoRa and GSM are two wireless technologies which we are considering using with our Smart Irrigation Controller. LoRa is a LPWAN (i.e. Low Power Wide Area) technology which can connect with other devices at a range of up-to 20KM, although it offers a very limited bandwidth. On the other hand, GSM is a popular wireless technology used by mobile operators to provide data services and internet connectivity. If we were to implement a LoRa module, we need to (1) select a LoRa module operating in a frequency band we can legally use in Lebanon (i.e. like ISM band 868MHz), (2) and setup a LoRaWAN gateway operating at the same frequency as the LoRa module. If we were to implement a GSM module, we require that (1) our GSM module supports the frequency bands used by Lebanese mobile operators for 2G/EDGE (i.e. 900MHz to 1.2GHz range), and (2) a SIM card from a Lebanese mobile operator with a valid subscription to connect to the closest GSM relay. A reference to the Lebanese National Frequency allocation Table can be found [here](#).

To implement LoRa and GSM support for the Raspberry-Pi, we designed an add-on PCB which supports these respective modules, and can be connected to the

Raspberry-Pi through its GPIO interface. Having both LoRa and GSM modules available simultaneously on the PCB would be redundant, as we only require one of these modules to ensure internet connectivity. As such, we plan on allowing the users to select which module they would prefer to use by toggling a dipswitch connected to the LoRa and GSM modules slots. Selecting either module with the dipswitch will modify which scripts will be executed on the Raspberry-Pi, effectively providing seamless support for communication when using either LoRa or GSM. Currently, our addon PCB supports the *MTXDOT-EU1-A00-1* LoRa module and the *NL-SW-GPRS* GSM module, which both communicate with the Raspberry-Pi through an UART serial interface. On a side-note, we have added additional pin headers on our addon PCB, to act as a bridge with the unused pins on the Raspberry-Pi's GPIO interface. This design choice has been made to ensure we would have an uncluttered interface in later stages of the project when we would work on connecting our Smart Irrigation Controller to physical irrigation systems.



*Figure 4 - MTXDOT-EU1-A00-1 (Left) and NL-SW-GPRS (Right)*

Our add-on board was supposed to follow the official design guidelines from the Raspberry-Pi foundation for “HATs” (The term HAT stands for ‘Hardware on Top’ and is

used to describe add-on board connecting to the GPIO interface of the Raspberry-Pi) but considering that we are still in a prototyping stage, we decided to go with a design which does not completely abide to these guidelines (i.e. No ID EEPROM and dimensions which differ from the recommended design). The official design guidelines for HATs can be found [here](#), and the recommended mechanical drawing is available [here](#).

To design our add-on PCB, we have used the electronic design automation software *KiCad*, which is open-source and contains all required tools to work on our design. These steps include (1) developing custom symbols and footprint for parts not available in stock libraries, (2) creating a schematic file containing the desired end circuit, (3) setting the dimensions for our PCB and placing & connecting our components, with a two-layer design in mind, (4) generating our BOM (i.e. Bill-of-Material) and a 3D model of our PCB to visualize the end result.

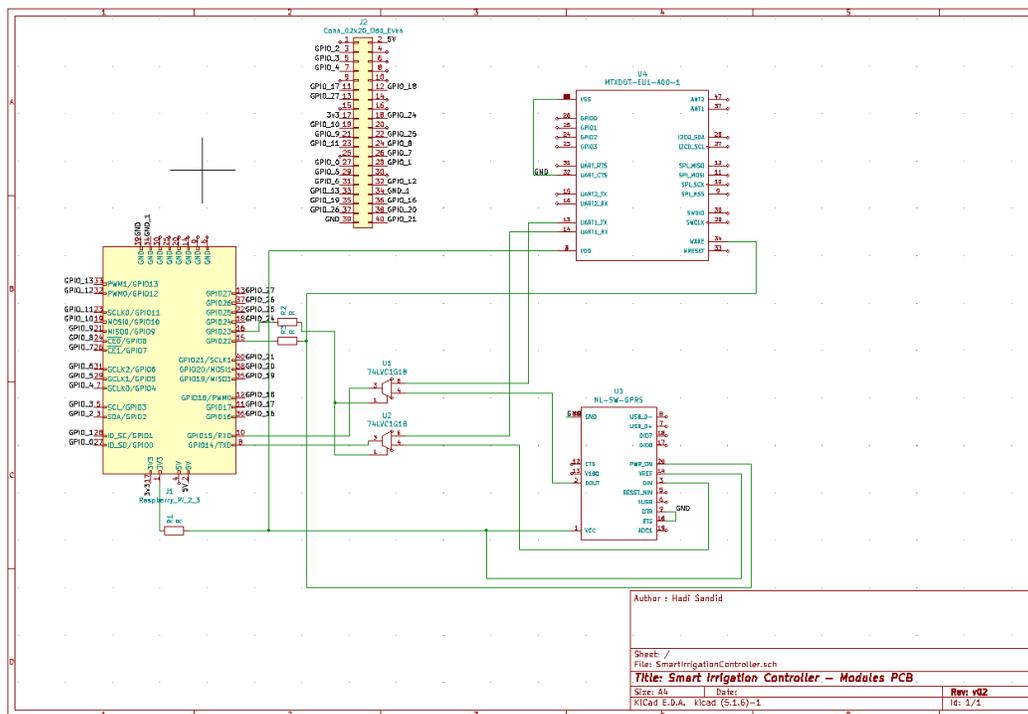
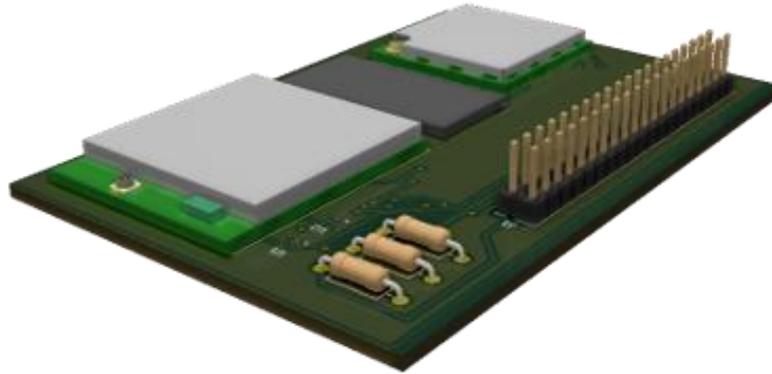


Figure 5- Schematic of our add-on PCB



*Figure 6- 3D Visualization of our add-on PCB*

## Back-End Server

The Back-End Server in the Smart Irrigation System was developed by Roaa Al-Feel, a graduate student assisting Dr. Saghir for the Smart Irrigation System project. It is based on Python, runs on the micro-web framework Flask, and stores user/field data as JSON files in a NOSQL database. RESTful API Calls for the back-end system are available in *APPENDIX F*.

The purposes of the Back-End Server include : (1) Computing efficient irrigation routines for agricultural fields in a set region, based on satellite data, evapotranspiration values, and other geographical & weather parameters; (2) Sending targeted information about optimal irrigation routines to Smart Irrigation Controllers, or to user interfaces connected with the system (i.e. Web Interface & Mobile Application); (3) Receiving information around completed irrigation routines, either automatically from deployed Smart Irrigation Controllers, or manually from users logged on either the Web Interface or Mobile Application.

Currently, the Back-End Server is hosted on the Google Cloud platform with limited scalability, which is enough for us to run tests, and ensure our Back-End Server coordinates properly with other components of the system. In the final stages of the Smart Irrigation System, we plan to remove scalability constraints on the Back-End Server, which will allow us to serve a larger userbase once access to the system is made public. Data from both users and their fields is also collected for analytics purposes. This data is used to study the preferences of our user-base and decide if we should later change some aspects of our system (i.e. UI, prediction systems...).

## Web Interface & Mobile Application

To allow our users to interact with the Smart Irrigation System, and receive data relevant to their fields, we had to make sure that our user portal would be easy-to-use and accessible from everywhere, on almost any device. Therefore, we decided to develop both a Web Interface and Mobile Application component for the Smart Irrigation System.

The purpose of the Web Interface is to allow users to re-configure or set preferences for their Smart Irrigation Controller, while also being able to check the irrigation history on any fields they have registered. The Web Interface UI has been designed using the Bootstrap front-end framework, and all logic and user interactions are handled through JavaScript. Whenever the user requests or sends any data, additional JavaScript code is executed as to handle proper communication between the device running the Web Interface and the device being contacted (i.e. Smart Irrigation Controller or Back-End Server). Initially, we planned on hosting our Web Interface directly on our Smart

Irrigation Controller using the Linux web server *Apache*, which would allow users to access it by just having the device they're browsing from be in the same Wi-Fi network as the Smart Irrigation Controller. However, we recognized it would cause issues in the long-term, as hosting the Web Interface on the Smart Irrigation Controller would mean it would have to operate without downtime to service possible requests from users, consuming more electricity and making it near-impossible to power the Smart Irrigation Source without connecting it to a power grid. An alternative we are considering is to set the Web Interface in our Back-End Server. This approach would allow users to access the Web Interface with near-zero downtime, and have easier access to their data, as it is already available in the Back-End Server's database. One downside to this approach might be that configuring Smart Irrigation Controller might require more steps, as the user now sends his requests to the Back-End Server, which must now forward it to the targeted Smart Irrigation Controller.

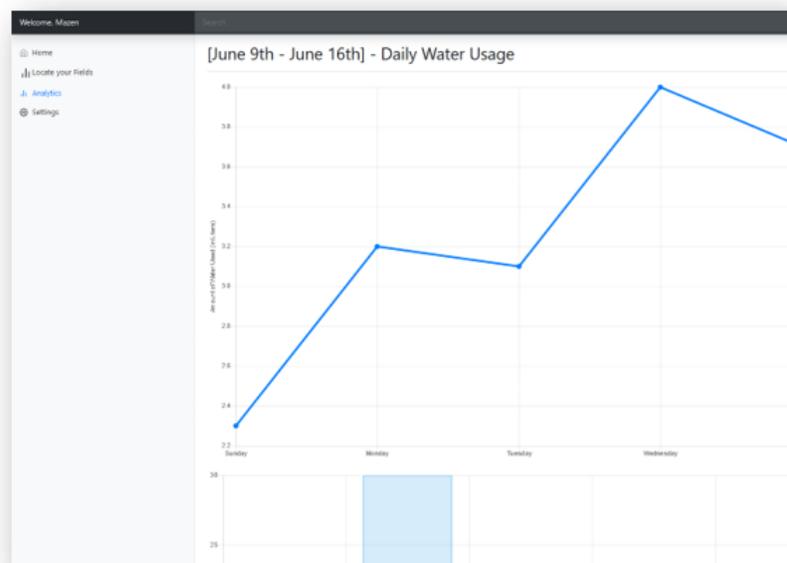


Figure 7- Web Interface, 'Analytics' page

The Mobile Application has a similar purpose to the Web Interface. Currently, only an Android version is available, but we plan to extend support to IOS devices soon. The Mobile Application has been designed to have an easy-to-use interface, multi-language support (Arabic/English are currently supported), and compatibility with devices running Android versions as old as *Android 4.0 Jellybean*.

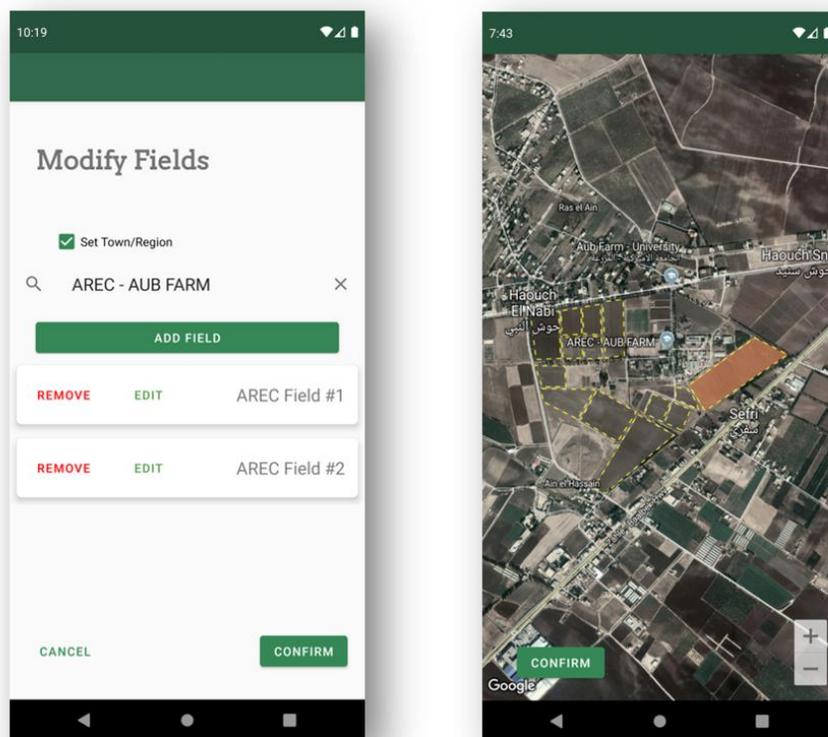


Figure 8- Mobile Application

## Documentation

To ensure the readability and re-usability of our code and hardware designs, we had to provide in-depth documentation. This is especially important considering that we would

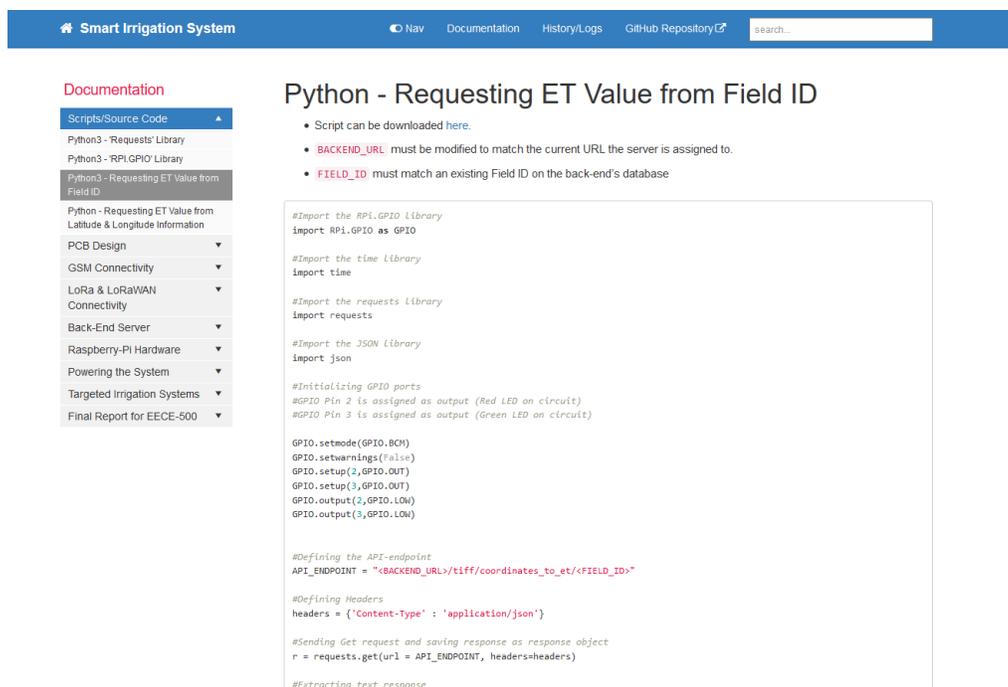
like to make our project open-source, and easily modifiable by any individual or third-party.

We decided to make our documentation publicly available online on a separate website.

To have a functional website, we have used the static site generator *Jekyll*, which runs on *Ruby* and generates a website from the markdown files we wrote for documentation.

We have also used the 'github-pages' functionality on *GitHub* to build and host our documentation website at no cost. The URL to our webpage is

<https://hsandid.github.io/SmartIrrigationSystem/>.



The screenshot shows a web browser displaying the 'Smart Irrigation System' documentation website. The top navigation bar includes 'Nav', 'Documentation', 'History/Logs', and 'GitHub Repository'. A search bar is located on the right. The left sidebar contains a 'Documentation' section with a dropdown menu listing various topics, including 'Python3 - Requesting ET Value from Field ID'. The main content area displays the title 'Python - Requesting ET Value from Field ID' and a list of bullet points: 'Script can be downloaded here.', 'BACKEND\_URL must be modified to match the current URL the server is assigned to.', and 'FIELD\_ID must match an existing Field ID on the back-end's database'. Below the text is a code block containing Python code for GPIO setup and an API request.

```
#Import the RPI.GPIO library
import RPi.GPIO as GPIO

#Import the time library
import time

#Import the requests library
import requests

#Import the JSON library
import json

#Initializing GPIO parts
#GPIO Pin 2 is assigned as output (Red LED on circuit)
#GPIO Pin 3 is assigned as output (Green LED on circuit)

GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
GPIO.setup(2,GPIO.OUT)
GPIO.setup(3,GPIO.OUT)
GPIO.output(2,GPIO.LOW)
GPIO.output(3,GPIO.LOW)

#Defining the API-endpoint
API_ENDPOINT = "<BACKEND_URL>/tiff/coordinates_to_et/<FIELD_ID>"

#Defining Headers
headers = {'Content-Type': 'application/json'}

#Sending Get request and saving response as response object
r = requests.get(url = API_ENDPOINT, headers=headers)

#Extracting text response
```

Figure 9 - Documentation Website

## Global, Economic, Environmental, and Societal Impact of Completed Work

As mentioned previously, the Smart Irrigation System's objective is to 'enhance' the irrigation process on agricultural fields, as to reduce water wastage and maximize crop yield. In the scope of a country like Lebanon, this project might help address economic, environmental, and societal issues. If we consider the economical aspect, a considerable amount of companies and individuals have started a shift towards agriculture-based ventures. Using the Smart Irrigation System might allow them to maximize their crop yield and ensure better returns on their businesses. If we consider the environmental aspect, the Smart Irrigation System delivers highly optimized irrigation routines, which aim for a near 0% of water wastage. This is particularly important considering how scarce water resources can be in certain regions of Lebanon. If we consider the societal aspect, the Smart Irrigation System can help alleviate some of the issues brought by the current Lebanese economic crisis. Lower-class families are losing most of their purchasing power and might soon not be able to buy food due to the rising prices of imports. The solution would be to increase local production to try and match the demand, but these efforts are hindered by the lack of an efficient irrigation solution in most agricultural fields. This is where we can deploy the Smart Irrigation System, to help optimize irrigation routines in these fields. Its use will lead to a higher crop yield, which will in turn increase local production of plants/fruits/vegetables. It is also in our plans to release the Smart Irrigation System as an open-source project, available for use and modification to all. This way, we can hope

custom iterations of our system are developed and fine-tuned to support additional components & address issues in other specific areas of the world.

## Conclusion

I have greatly benefited from this training experience. It has allowed me to develop technical skills, thanks to both extensive research and advice from my supervisor Dr. Saghir.

Technical skills : (1) Multidisciplinary R&D, focusing mainly on hardware and software development, (2) Write scripts in Python to control the Raspberry-Pi's interfaces, and different aspects of a Linux System, (3) Working from scratch on a complete PCB design, using the open-source tool Ki CAD, (4) Using front-end frameworks like Bootstrap, and static site generators like Jekyll, to design user-interfaces and documentation, (5) Learning more about Embedded Systems, mainly by experimenting with the Raspberry-Pi, (6) Learning more about RF & EM considerations in PCB design, (7) Learning more about alternative wireless solutions for data services, like LoRa and GSM.

Due to the limited background I have in some topics, I had to ask for advice from Dr. Saghir and other professors in MSFEA (on Dr. Saghir's behalf ) when working on some tasks. One example would be the issues I encountered while modifying our add-on PCB to take into consideration RF design constraints. This led to issues when researching RF & EM interference, as I only knew the basics on this topic from my Engineering Electromagnetics course (EECE-380). Dr. Joseph Costantine, whose field of interest is

RF & EM, agreed to answer some of my questions on the topic, which allowed me to complete the task successfully. He also recommended taking a course on Radio Frequency (RF) Circuits Design (EECE-685) if I wanted to learn more about the topic, or if I intended to work on RF circuits in future projects.

I have no improvement suggestion for the training program. Throughout the course of the internship, I was well guided by Dr. Mazen and all tasks were set with clear goals.

## References

- [Official Request Library documentation](#)
- [Official RPI.GPIO Documentation](#)
- [Altium - PCB Design Recommendations](#)
- [Electronics-Notes - PCB Design Recommendations](#)
- [Quick-PCBA - PCB Design Tips](#)
- [Ken's Tech Tips - Cellular Networks Technologies](#)
- [LoRa-Alliance Official Documentation](#)
- [The Things Network - Commercially Available LoRaWAN Gateways](#)
- [Raspberry-Pi 3 Model B, Official Page](#)
- [Raspberry-Pi Zero, Official Page](#)
- [Raspberry-Pi, Github - Pi-HATs documentation](#)
- [PiHut - How do I power my Raspberry Pi?](#)
- [HowToForge - What is Incron ?](#)
- [SSWM - Drip Irrigation](#)
- [SSWM - Sprinkler system](#)
- [SSWM - Surface Irrigation](#)
- [Github Repository for the Smart Irrigation System](#)

# Appendices

## Appendix A: Background Information on the Project

This Smart Irrigation System was originally submitted by a team of professors at AUB for the *Google AI Impact Challenge 2019*, whose objective was to propose projects making use of AI to address societal issues. The Smart Irrigation System was among the projects selected by Google, effectively securing guidance in further steps of project development.

The team of AUB professors is composed of Dr. Hadi Jaafar, which lead the project and was responsible of the smart irrigation component; Dr. Fatima Abu Salem, which was responsible of the machine-learning and AI components; Dr. Mazen Saghir, which lead the IoT and embedded systems components; and Dr. Samer Kharroubi, which lead the analysis and statistical modeling components.

More information about the project can be found here :

<https://www.aub.edu.lb/msfea/news/Pages/googleai-impactchallenge.aspx>

My involvement in the project started in August 2019, where I have assisted Dr. Mazen with the development of an Android mobile application, which would be part of the Smart Irrigation System. Throughout both the Fall and Spring semesters, I have continued my work and assisted with the development of other components in the system, both through the work study program and by taking EECE-499 with the Smart Irrigation System as my research subject. This current summer internship, which fulfills the requirements of EECE-500, is a continuation of my work on the project.

## Appendix B: Raspberry-Pi Power Requirements Overview

The Raspberry-Pi can be powered in two ways :

- (1) Through its Micro-USB port, with a recommended input voltage of 5V input current of 2A



Figure 10 - Micro-USB port of the Raspberry-Pi

- (2) Through it's GPIO interface by plugging a 5V source to Pin #2 on the GPIO header, and the ground of this 5V source to Pin #6 on the GPIO header.

Pin#	NAME		NAME	Pin#
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , PC)		DC Power 5v	04
05	GPIO03 (SCL1 , PC)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (PC ID EEPROM)		(PC ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

Figure 11- GPIO Pins used to Power the Raspberry-Pi

Warning : The recommended method of powering the Raspberry-Pi is through its Micro-USB port, as it offers regulation and fuse protection to protect from over-voltage and current spikes. **There is no regulation and fuse protection on the GPIO Interface** meaning that any over-voltage, current spikes, or reverse polarization might fry the GPIO interface, or worse, the Pi itself.

Source: [PiHut - How do I power my Raspberry Pi?](#)

## Appendix C: Raspberry-Pi 3 Model B Overview



Figure 12- Raspberry-Pi 3 Model B

Processor: *Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz (Quad-core processor).*

Networking: *Dual-band wireless LAN / 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac Wireless LAN; Bluetooth 4.2/BLE (Bluetooth Low-Energy); Ethernet port; Power-over-Ethernet support (Requires separate module)*

RAM: *1GB LPDDR2 SDRAM*

GPIO Interface:

- *Extended 40-pin GPIO header; GPIO I/O pins can set/read at high (3.3V) or low (0V); Internal pull-up or pull-down resistors are present on each pins. Pins*

*GPIO2-GPIO3 have fixed pull-up resistors, but other pins can be configured in software.*

- *Raspberry-Pi supports Pulse-Width modulation, with Software Pulse-Width Modulation is available on all GPIO pins, and Hardware Pulse-Width Modulation available on pins GPIO12, GPIO13, GPIO18, GPIO19.*
- *Raspberry-Pi supports UART with the following configuration: RX(GPIO15), TX (GPIO16),*
- *Raspberry-Pi supports SPI with the following configuration: SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7); SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16).*
- *Raspberry-Pi supports I2C with the following configuration : EEPROM Data: (GPIO0); EEPROM Clock (GPIO1); Data: (GPIO2); Clock (GPIO3).*

Other Interfaces: *4x USB 2.0 ports; CSI camera port for connecting a Raspberry Pi camera; DSI display port for connecting a Raspberry Pi touchscreen display; 4-pole stereo output and composite video port; Micro SD port for loading your operating system and storing data; 1x Full-size HDMI port*

Required Power: *Typical PSU current capacity is 2.5A; Typical bare-board active current consumption is 400ma; Maximum total USB peripheral current draw is 1.2A; Recommended micro-USB connector: 5 V/2.5 A DC*

Miscellaneous Information: *Raspberry-Pi 3 Model B will remain in production until January 2026; Global compliance and safety certificates [here](#); Operating temperature : 0-50°C; Raspberry-Pi 3 Model B mechanical drawing available [here](#); Raspberry-Pi 3 Model B schematic diagrams available [here](#)*

Source: [Raspberry-Pi 3 Model B, Official Page](#)

## Appendix D: Python3 'Requests' Library Overview

### Basic requests

- Importing the 'requests' library

```
import requests
```

- GET request

```
requests.get(<URI>)
```

- POST request

```
requests.post(<URI>,<DATA>)
```

- PUT request

```
requests.put(<URI>, <DATA>)
```

- DELETE request

```
requests.delete(<URI>)
```

### Adding custom parameters

- GET request with custom parameters

```
payload = {'key1': 'value1', 'key2': 'value2'}  
r = requests.get(<URI>, params=payload)
```

### Adding custom headers

- GET request with custom headers

```
url = 'https://api.github.com/some/endpoint'  
customHeaders = {'user-agent': 'my-app/0.0.1'}  
r = requests.get(<URI>, headers = customHeaders)
```

### Read received data

- GET request with a JSON response

```
r = requests.get('https://api.github.com/events')  
jsonResponse = r.json()  
print(jsonResponse[<JSONFIELD>])
```

### Source:

- [Official Request Library documentation](#)

## [Appendix E: Python3 'RPI.GPIO' Library Overview](#)

### Initialization:

- Importing the 'RPI.GPIO' library

```
import RPi.GPIO
```

- Select a pin numbering system

```
#BOARD numbering system  
GPIO.setmode(GPIO.BOARD)
```

```
#BCM numbering system  
GPIO.setmode(GPIO.BCM)
```

Configure channel as an input:

```
#With Pull-up resistor  
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

```
#With Pull-down resistor  
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

- Configure channel as an output:

```
GPIO.setup(channel, GPIO.OUT)
```

I/O configuration:

- Poll GPIO pins

```
if GPIO.input(channel):  
    print('Input was HIGH')  
else:  
    print('Input was LOW')
```

- Input detection using *WaitForEdge()* function

```
GPIO.wait_for_edge(channel, GPIO.RISING)
```

- Set GPIO pin as high/low

```
#Set output as high  
GPIO.output(<CHANNEL>, GPIO.HIGH)
```

```
#Set output as low  
GPIO.output(<CHANNEL>, GPIO.LOW)
```

### Clean-up:

- To clean-up any used resources, you can call the following :

```
GPIO.cleanup()
```

### Source:

- [Official RPI.GPIO Documentation](#)

## Appendix F: Back-End API Calls

Disclaimer: Our back-end is still under testing. As such, an URI will not be made publicly available.

### Summary of API Calls :

- POST [/signup](#)
- POST [/login](#)
- POST [/fields](#)
- GET [/fields](#)
- GET [/fields/<id>](#)
- PUT [/fields/<id>](#)
- GET [/irrigation](#)
- GET [/irrigation/<irrigation\\_id>](#)
- PUT [/irrigation/<irrigation\\_id>](#)
- GET [/tiff/last\\_updated](#)
- POST [/app\\_logs/device](#)
- POST [/app\\_logs/app\\_lang](#)
- POST [/app\\_logs/device\\_lang](#)
- POST [/app\\_logs/region](#)
- POST [/app\\_logs/exit\\_screen](#)
- POST [/app\\_logs/ui\\_errors](#)
- POST [/app\\_logs/server\\_errors](#)
- POST [/app\\_logs/lang\\_change](#)
- POST [/app\\_logs/ui\\_response](#)

### POST /signup :

- Description:

*Call allowing new users to sign-up and receive an authentication token/user-id.*

- Request body (in *JSON* format) :

```
{  
  mobile_number: "+961<phoneNumber>",  
  town_id: "<townID>"  
}
```

- Response body (in *JSON* format) :

```
{  
  user_id: "<userID>",  
  auth_token: "<auth_token>"  
}
```

#### POST /login :

- Description:

*Allows users to log-in based on a verification code. Currently not included in features.*

- Request body (in *JSON* format) :

```
{  
  mobile_number: "<mobileNumber>",  
  verification_code: "<verificationCode>"  
}
```

- Response body (in *JSON* format) :

```
{  
  user_id: "<userID>",  
  auth_token: "<authToken>"  
}
```

#### POST /fields :

- Description:

*Call for users to create a new field*

- Required Header : auth\_token

- Request body (in JSON format) :

```
{
  name: "<fieldName>",
  area: "<areaID>",
  latitude: <number>,
  longitude: <number>,
  irrigation_system_id: "<irrigationSystemID>",
  crop_type_id: "<cropTypeID>",
  town_id: "<townID>"
}
```

- Response body (in JSON format) :

```
{
  field_id: "<fieldID>"
}
```

### GET /fields :

- Description:

*Returns all fields registered by a specific user*

- Required Header : auth\_token
- Response body (in JSON format) :

```
{
  fields:
  [
    {
      name: "<fieldID>",
      area: <number>,
      latitude: <number>,
      longitude: <number>,
      crop_type_id: "<cropTypeID>",
    }
  ]
}
```

```
        Irrigation_system_id: "<irrigationSystemID>",
        town_id: "<townID>",
        user_id: "<userID>",
        created_at: <timestamp>,
        updated_at: <timestamp>
    }
]
```

GET /fields/<id>:

- Description:

*Returns a specific field which belong to the current user*

- Required Header: auth\_token
- Response body (in JSON format) :

```
{
  name: "<fieldID>",
  area: <number>,
  latitude: <number>,
  longitude: <number>,
  crop_type_id: "<cropTypeID>",
  Irrigation_system_id: "<irrigationSystemID>",
  town_id: "<townID>",
  user_id: "<userID>",
  created_at: <timestamp>,
  updated_at: <timestamp>
}
```

PUT /fields/<id> :

- Description:

*Updates a field created by the current user*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{
  name: "<fieldID>",
  area: <number>,
  latitude: <number>,
  longitude: <number>,
  crop_type_id: "<cropTypeID>",
  Irrigation_system_id: "<irrigationSystemID>",
  town_id: "<townID>",
  user_id: "<userID>",
  created_at: <timestamp>,
  updated_at: <timestamp>
}
```

#### GET /irrigation:

- Description:

*Returns the irrigation schedule/history for a field*

- Request body (in JSON format) :

```
{
  field_id: "<fieldID>",
  scheduled: <boolean>
}
```

- Response body (in JSON format) :

```
{
  field_id: "<fieldID>",
  watered: <boolean>,
  duration_needed: <number>,
  actual_duration: <number>,
  water_needed: <number>,
  actual_water_used: <number>,
  scheduler_irrigation_time: <timestamp>,
  actual_irrigation_time: <number>,
  current_fuel_price: <number>,
  actual_fuel_cost: <int>,
}
```

```
emissions: <int>,
created_at: <timestamp>,
updated_at: <timestamp>
}
```

#### GET /irrigation/<irrigation\_id> :

- Description:

*Returns the irrigation schedule/history for a field*

- Response body (in JSON format) :

```
{
  field_id: "<fieldID>",
  watered: <boolean>,
  duration_needed: <number>,
  actual_duration: <number>,
  water_needed: <number>,
  actual_water_used: <number>,
  scheduler_irrigation_time: <timestamp>,
  actual_irrigation_time: <number>,
  current_fuel_price: <number>,
  actual_fuel_cost: <number>,
  emissions: <number>,
  created_at: <timestamp>,
  updated_at: <timestamp>
}
```

#### PUT /irrigation/<irrigation\_id> :

- Description:

*Call which allows the user to notify that he irrigated one of his fields*

- Request body (in JSON format) :

```
{
  field_id: "<fieldID>",
  watered: <boolean>,
}
```

```
actual_duration: <number>,
actual_water_used: <number>,
actual_irrigation_time: <number>,
actual_fuel_cost: <number>,
emissions: <number>,
created_at: <timestamp>,
updated_at: <timestamp>
}
```

#### GET /tiff/last\_updated :

- Description:

*Call to get the time at which the TIFF map was last updated*

- Response body (in JSON format) :

```
{
  last_updated: <Datestring>
}
```

#### POST /app\_logs/device :

- Description:

*Get user device model. This will be stored for each user, overwritten at each request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{
  device_model: "<deviceModel>"
}
```

- Response:

status 204 [NO\_CONTENT]

### POST /app\_logs/app\_lang :

- Description:

*Get language to which the application is currently set. This will be stored for each user, overwritten at each request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  app_lang: "<appLanguage>"  
}
```

- Response:

status 204 [NO\_CONTENT]

### POST /app\_logs/device\_lang :

- Description:

*Get language to which the user device is currently set. This will be stored for each user, overwritten at each request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  device_lang: "<deviceLanguage>"  
}
```

- Response:

status 204 [NO\_CONTENT]

### POST /app\_logs/region :

- Description:

*Get the current region of the user. This will be stored for each user, overwritten at each request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  region: "<regionName>"  
}
```

- Response:

```
status 204 [NO_CONTENT]
```

#### POST /app\_logs/exit screen :

- Description:

*Get the screen from where the user exits the application the most frequently. This will be stored for each user, overwritten at each request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  screen_name: "<screenName>"  
}
```

- Response:

```
status 204 [NO_CONTENT]
```

#### POST /app\_logs/ui\_errors :

- Description:

*Periodically check if any UI error occurred when using the application. This will be stored on a system-level. Contents of this list will be added to the list of errors already stored in database*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  ui_errors:
```

```
[
  {
    error_msg: "<errorMsg>",
    time: <timestamp>
  }
]
```

- Response:

```
status 204 [NO_CONTENT]
```

#### POST /app\_logs/server\_errors :

- Description:

*Periodically check if any server error occurred when using the application. This will be stored on a system-level. Contents of this list will be added to the list of errors already stored in database*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{
  server_errors:
  [
    {
      error_msg: "<errorMsg>",
      time: <timestamp>
    }
  ]
}
```

- Response:

```
status 204 [NO_CONTENT]
```

#### POST /app\_logs/lang\_change :

- Description:

*Check if the user has changed the application language manually. This will be stored on a user-level. It will be added to the stored total value, please reset to zero locally after sending request*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  lang_change: <number>  
}
```

- Response:

status 204 [NO\_CONTENT]

#### POST /app\_logs/ui\_response :

- Description:

*Get the average UI response time for each user. This will be stored on a user-level. It will be added to the stored total value and a counter of total numbers this response time has been logged will be incremented. Therefore calculating average response time per user.*

- Required Header : auth\_token
- Request body (in JSON format) :

```
{  
  response_time: <number>  
}
```

- Response:

status 204 [NO\_CONTENT]

## [Appendix G: Raspberry-Pi – Power Consumption Benchmarks](#)

We know that the Raspberry-Pi requires an input voltage of 5V. We will now see how much current (and coincidentally power) some models of the Raspberry-Pi draw under

different load conditions. All the benchmarks will be taken from the *PiDramble* website.

The boards are running stock Raspbian Lite, with no additional software installed and only running basic daemons. There are no additional peripherals connected to the Pi boards.

Note: All Raspberry-Pi models consume around 0.1W when powered off, until they are disconnected from their power source.

Raspberry-Pi Model	Pi State	Power Consumption
<b>Raspberry-Pi 3 B+</b>	Idle	350mA(1.9W)
<b>Raspberry-Pi 3 B+</b>	Maximum CPU Load	980 mA(5.1 W)
<b>Raspberry-Pi 3 B+</b>	Minimum CPU Load with HDM & LED disabled	350 mA(1.7 W)
<b>Raspberry-Pi 3 B</b>	Idle	260 mA(1.4 W)
<b>Raspberry-Pi 3 B</b>	Maximum CPU Load	730 mA(3.7 W)
<b>Raspberry-Pi 3 B</b>	Minimum CPU Load with HDM & LED disabled	230 mA(1.2 W)
<b>Raspberry-Pi 2 B</b>	Idle	220 mA(1.1 W)
<b>Raspberry-Pi 2 B</b>	Maximum CPU Load	400 mA(2.1 W)
<b>Raspberry-Pi 2 B</b>	Minimum CPU Load with HDM & LED disabled	200 mA(1.0 W)

*Figure 13- Power requirements for different Raspberry-Pi models*