Università
della
Svizzera
italiana

**Software
Institute**

GENIE

A Physical Way to Visually Represent Data

Francesco Simone Arrigo

Supervised by
Prof. Dr. Michele Lanza

Cosupervised by
Dr. Marco D'Ambros
Dr. Csaba Nagy

SOFTWARE & DATA ENGINEERING MASTER THESIS

Abstract

Visual analytics platform are very useful to display data and information in a reasonable and understandable way so that users may infer useful aspects of the data.

Unfortunately, they usually depict the underlying data using sophisticated interactive dashboards, which often introduce a considerable accidental complexity. However, end users are not always interested in spending their time manipulating the represented data, but rather care only about a specific and intrinsically simple aspect of it.

Our goal is to leverage recent advances in cyber-physical devices that can physically represent the status and eventual changes of a particular aspect of the data. Such devices include lights whose color and intensity can be modified programmatically.

Our thesis project revolves around the development of a toolset which can distill relevant aspects of many kind of underlying data into simple signals which can then be manifested using said lights, thus bridging the gap between the data space and the physical world.

Shoot for the moon.
Even if you miss,
you'll land among the stars.

-Norman Vincent Peale

Acknowledgements

Despite how much I would like to think that I could do everything without relying on anyone else the truth is that I would not have come so far without the help of so many amazing people

First and foremost to Professor Michele Lanza , Csaba Nagy, Marco D'Ambros and Roberto Minelli for their apparently close to infinite amount of patience they have shown me during this work

To Professor Lanza in particular because of his incredible determination, attitude and "everything can work" mentality that has helped me grow so much in these years

To all the professors at USI, it is incredible how many awesome professors and people are present in this faculty

To all my classmates both current and past for the countless sleepless nights working on something together and making each other smile

To all my friends, both inside and outside school, that were there to bring me up when I was feeling down. Without them I would have probably left USI a long time ago

To all the people that will not fit in this page, there are far too many to list them all

To all the people who are still here and to the ones that are not here anymore

To all of you

Thank you, from the bottom of my heart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dominant way to display and reason about data is to use interactive or static dashboards. This approach is very effective in most cases but can pose problems in others [18].

For example, these platform often hide some critical aspects of the data to make the visualisation more appealing. This is a dangerous double-edged sword since a user may overlook critical information that would have been more visible had the representation been different. For example it is particularly costly and time consuming in the DevOps [15] world where new features and updates need to be delivered in a short timespan and not being able to quickly react to relevant aspects of the available data might lead to problems that could slow down the whole development and deployment process [14].

These types of interruptions are called as "in-person interruptions". Researchers often considered them more costly and dangerous than the so-called "computer-based interruptions," which can be mitigated through strategies, for example, automatically detecting a worker's interruptibility.

As a possible way to tackle such interruptions, Shepherd et al. proposed FlowLight [21]. FlowLight, combines a physical traffic-light like LED, with an automatic interruptibility measure based on computer interaction data. They studied 446 participants from 12 countries and found that a simple approach like FlowLight, can reduce the interruption by 46% of a worker and increase their awareness of potential disruptiveness of interruptions.

The study of Shepherd et al. and the simpleness of FlowLight motivated our work to investigate and develop a more general approach. While FlowLight is dedicated to displaying with an RGB LED if a user is free to talk or busy, we aim at implementing the concept of using a physical device like an LED to convey information to the user in a more general way to represent various information that may be difficult to show on a usual dashboard.

1.1 Thesis Goal

With our thesis we aim at alleviating the described problem by representing particular aspects of the data in a simpler yet effective way. In particular our approach is focused around the use of various physical devices like one or multiple RGB lights to show changes and the status of data.

Our work is meant to provide an easy to expand system that will allow developers to represent data with one or multiple RGB LEDs so they could react and reason about data in a faster

and simpler manner, with this goal in mind we created an application that would be easy to expand thanks to a simple plugin system.

Physical devices are usually employed to show really simple information like notifications or on/off signals, our goal is to change this by developing a set of streamlined libraries to connect data, that may reside on a machine or coming from a stream of information, with devices so that users could visualise the data in a simple, physical way.

We chose RGB LEDs for this purpose given their huge array of possible representations when considering all the different color combinations paired with the possibility of having various effects like blinks and simple animations.

1.2 Structure of the Document

The thesis is organized as follows:

- Chapter 2 contains a brief introduction to our application, called Genie, and some related work.
- Chapter 3 is dedicated to the process we followed while choosing the hardware devices most appropriate for our work.
- Chapter 4 outlines the development of Genie (separately discussing its back and front ends) along with its execution flow.
- Chapter 5 is dedicated to present some example use cases that we created to show the potential of our concept.
- Chapter 6 summarizes our work and outlines possible future developments.

Chapter 2

A Physical Way to Visually Represent Data

In this chapter we introduce Genie and discuss the related work.

2.1 Overview

With Genie we provide a simple to expand applicaition that would be used to represent data with simple RGB LEDs, in particular our vision was to create and implement a system that could take many different types of data.

In the simplest usage scenario, data would be translated into boolean operators signaling a light to turn on or off. However, we designed Genie in a more general way to use also various colors and different effects to have many more representation options and not be limited by only two different states.

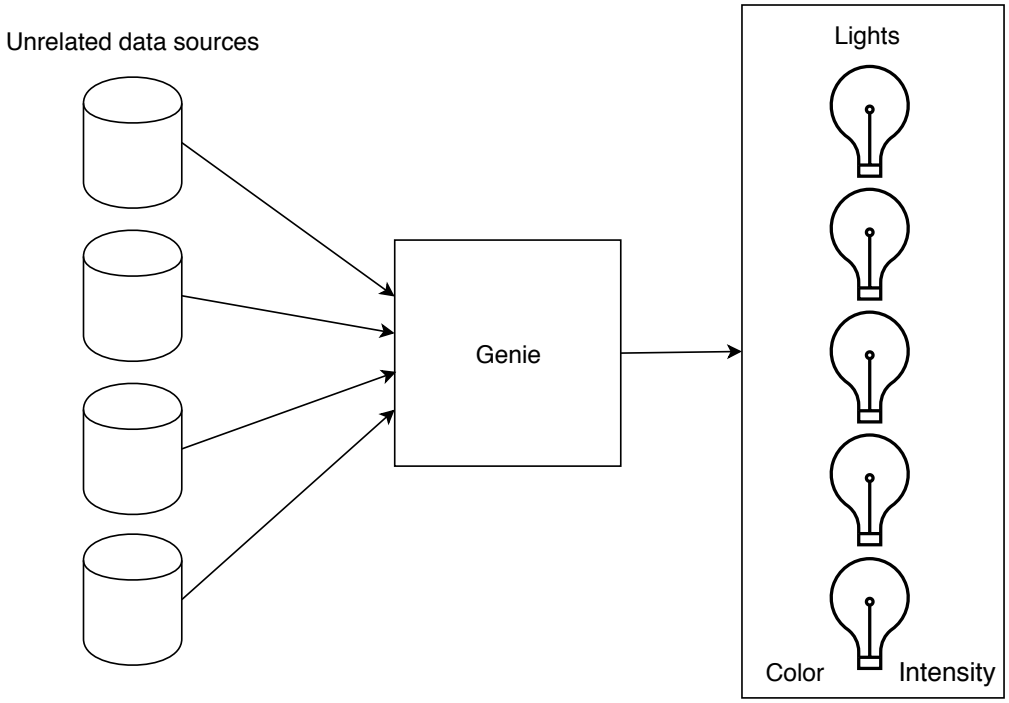

FIGURE 2.1: Genie translates data to visual effects in order to help users in observing critical information

Unfortunately we cannot make a system that will work with any kind of data, this is why it is really important for us to make sure the system is easy to expand so that users may customize it to better fit their needs. Nonetheless, we will provide some example applications that use many different type of data coming from different sources to highlight the flexibility of this approach. These examples include but are not limited to Google Calendar data, weather data and system data and will be discussed and described in detail in a dedicated chapter later.

The main design and implementation challenges of Genie can be summarized into six different points:

- Generalizability: supporting *any* kind of data from various data sources.
- Configurability: provide a general way to configure Genie in order to have different reactions to diverse types of data.
- Portability: being able to support multiple devices and platforms.
- Performance: Genie must be designed with very quick response time and be able to react quickly to changes in its input data stream .
- Efficiency: Genie has to use minimal resources and energy. In particular, it has to be able to run on low-resource devices such as an Arduino or RPi.
- Modifiability: It should be easy to add new functionalities to suit a particular user's needs.

2.2 Related Work

Our work can be categorized under the areas of data visualization and data reaction. In essence, visualization of complex data sources is a challenging task and many studied such approaches [13]. To better understand the background of Genie, we elaborate on two studies that are the most relevant in our field, and also inspired our work.

2.2.1 FlowLight

As we stated before, our work was inspired by the work of David Shepherd et al. on FlowLight [21] and the research conducted about it. This work is also what inspired us to use our device to represent important data in such a way that would not interrupt the work of the user.

FlowLight is a device that, thanks to a computer application, automatically monitors various aspects of the interactions between a user and the computer to detect is said user is free to be interrupted or is best left alone. It achieves this feat thanks to a system developed over the course of a year that continuously keeps track of various aspects of the computer's usage.

The interruption status is conveyed to co-workers thanks to a LED that used the same color scheme as a normal traffic light, this means that the status of a worker can immediately be seen and it is easy to understand thanks to the use of a familiar and universally accepted color scheme.

FIGURE 2.2: FlowLight in use

A study done on the application of FlowLight in a real world work environment [21] demonstrated that LED lights can be used to communicate information in a quick and effective way when paired to an easy to understand and agreed upon color scheme.

2.2.2 IF This Then That (IFTTT)

IF This Than That, usually abbreviated to IFTTT, is a web-based service that allows users to create some "triggers" that will execute one or multiple actions when triggered [20], these rules

are called "applets". Despite its inherent simplicity, this approach has been revealed to be very powerful since it can be applied to virtually any kind of problem.

Some example applets present on the website range from a simple reminder to drink water when a set amount of time has passed, to a notification that shows when the International Space Station is above the user's current location.

Such applets can be created with a simple and intuitive UI that allow developers to set triggers and actions to be executed without needed to code anything, this makes it very approachable to professional developers and normal users alike. The usage in which IFTTT shines is the automation of repetitive and/or mundane tasks like uploading the same photo to multiple social networks to reach a wider audience, by creating an applet that reacts to a new photo being uploaded to a specific website it is possible to trigger an event that will have said photo posted to all selected social networks at once instead of having to do it manually for each.

IFTTT is not only limited to the digital space however, with compatible devices it is possible to, for example, turn on the lights of a room when movement is detected. This makes it an effective method to bridge the gap between digital and physical world in a way similar to what we want to achieve.

Chapter 3

Device and platform choice

In this chapter we will discuss the various development options that we considered and discuss the reasons for our final decisions.

3.1 Overview

While choosing the devices and platform to use we had various decisions to make, in particular we needed to settle on an LED device that was neither too expensive to purchase nor too complicated to connect to a computer, and a platform to control said device that would be suitable for our needs.

In addition to the LEDs we needed to choose which programming language to use while developing the application to control them.

After some research we narrowed the decision to the following two platforms and four devices (in no particular order):

Devices

- Generic addressable RGB LEDs
- Blink(1)
  https://blink1.thingm.com/
- BlinkStick
  https://blinkstick.com/
- Philips HUE
  https://www2.meethue.com

Platforms

- Raspberry Pi
  https://www.raspberrypi.org/
- Arduino
  https://www.arduino.cc/

3.2 Devices

In order to select the devices that are the most suitable for our purposes, here we sum up the advantaged and disadvantages of each potential device we initially considered.

3.2.1 Generic addressable RGB LEDs

Pros:

- Really cheap to buy
- Customizable LED color and brightness

Cons:

- Hard to set up
- Hard to connect to a computer

A generic addressable RGB LED would be the cheapest way to implement our idea, the main problem with this device is that they usually come in long strips with a proprietary remote to control the color.

This is perfect for simple applications like ambient lights but not suitable for our implementation. In particular if we wanted to reduce the length of the strip it would be then necessary to solder a new connector to the new piece. To satisfy our generalizability requirement, i.e., to have an easily expandable system, we decided to avoid the inconvenience of soldering every new LEDs.

3.2.2 Blink(1)

Pros:

- Customizable LED color and brightness
- USB connection

Cons:

- LED placement on the device
- LED brightness

Blink(1) is a widely used RGB LED light for notification purposes [2]. This device was considered due to it being USB powered and having customizable brightness settings, this means that it could easily be connected to a computer and we would have had more options regarding the data representation.

What made us consider other devices instead of immediately settling on this one was the fact that the maximum brightness of this LED light was not really strong, this meant that it would have been hard to see it in the daytime or in nicely lit rooms. In addition to this the lights position on the device and the male USB type A connector meant that it would have been really hard to mount the device in a nice way for some applications.

3.2.3 Philips HUE

Pros:

- Customizable LED color and brightness
- Very high build quality

Cons:

- Lightbulb outlet connection
- Too big for our purpose

The HUE is a brand of LED lights by Philips, we initially considered these lights as an alternative due to their very high build quality and power.

Despite this, though, we decided to discard this options due to a multitude of factors. The biggest problem for our application was the fact that HUE lights are designed to be used as main or ambient lights for a room, this meant that they receive power from a standard lightbulb outlet and generally have the same size as a normal lightbulb. This meant that we could not easily connect it to a computer, this paired with its inherit size dictated by the need of being connected to a lightbulb outlet made us discard this option.

3.2.4 Blinkstick

Pros:

- Customizable LED color and brightness
- USB connection
- Open source hardware and software

Cons:

- Few programming languages supported
- LED placement on the device

Our final option was one, or multiple, devices offered by Blinkstick. Blinkstick offers 4 different devices to choose from: Blinkstick, Blinkstick Nano, Blinkstick Square and Blinkstick Strip.

TABLE 3.1: Blinkstick Devices

Product	LED Number	Max Brightness	Max Power Draw
Blinkstick	1	0.15 W	77 mA
Blinkstick Nano	2	0.30 W	137 mA
Blinkstick Square	8	2.00 W	500 mA
Blinkstick Strip	8	2.00 W	500 mA

Due to their much higher brightness level, the most suitable devices for our purposes are the Square and/or the Strip [3], the different LEDs arrangement of the two devices would also allow us to explore various methods to represent data.

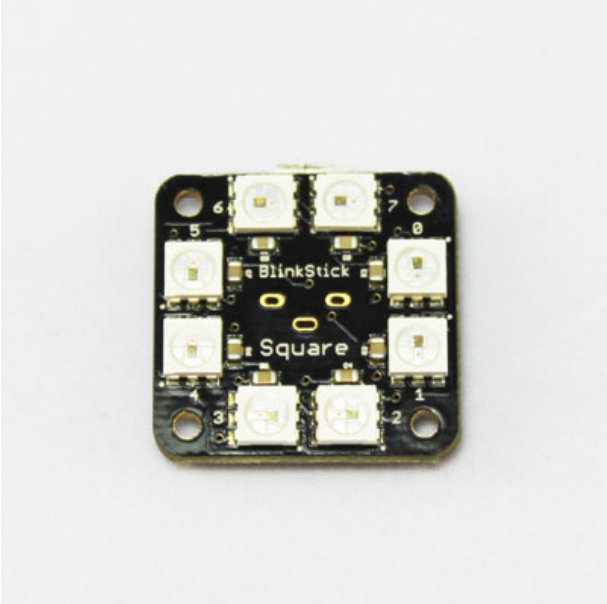

FIGURE 3.1: Blinkstick Square

1

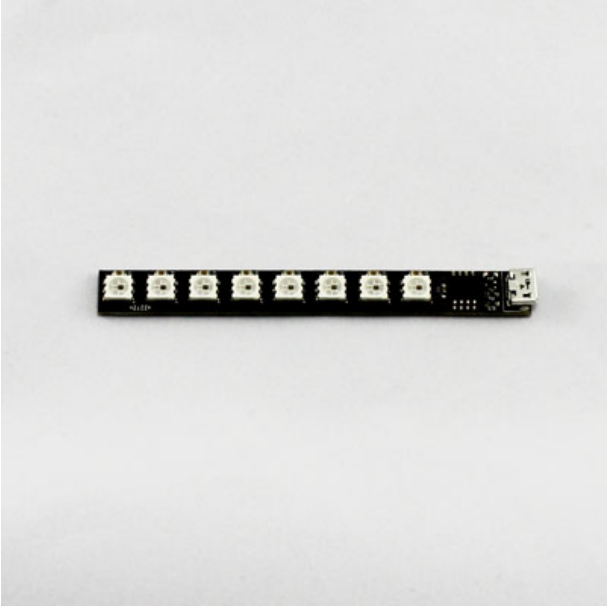

FIGURE 3.2: Blinkstick Strip

2

An important advantage of the Blinkstick devices is that there are eight LEDs on a single device, this means that the maximum brightness possible is really high (2W [3]) and it is possible to create simple animations even by only having a single device. In addition to this the open source nature of the device means that it is easier to figure out how the device works to build something new.

The main cons of this device are the limited selection of possible programming languages that can be used to control it and the LED placement. In particular, all the LEDs are placed on the same side of the device, this means that it is important to consider the final orientation of it in the final product to make sure the light would be easily visible.

3.3 Platforms

In addition to connecting the LEDs directly to a computer we demonstrate the usage of Genie through multiple examples of standalone applications. This due to the fact that we want our device to be as independent from a particular system as possible.

This means that a solution not requiring a dedicated PC would be ideal since it would make it more portable and easier to place in a spot that would make sense (since the light obviously has to be visible).

For this purpose we considered these two platforms to which the selected device could be connected.

3.3.1 Arduino

Pros:

- Small size
- Personal experience

Cons:

- Is a microcontroller

Arduino Uni is a potential candidate for the controller device of Genie. This choice was dictated by the small form factor of the platform which meant it would have been easy to hide from sight.

Unfortunately an Arduino is not really meant for this type of applications. Since the Arduino is a microcontroller board it is not suitable for our purpose of having an easily scalable and expandable system, all in all it could have worked but it would have been unnecessarily complicated to use this particular platform to implement our application.

3.3.2 Raspberry Pi

Pros:

- Small size
- Full fledged Linux-based computer

Cons:

- Some models have very limited resources

Another alternative alternative is to use a Raspberry Pi to control the attached devices. In addition to the small form factor, similar to the Arduino, that allows the Raspberry Pi to be

hidden from sight the upside of this approach is that a single solution will generally work both on a dedicated computer and on the Raspberry Pi.

The main downside of this approach is that, depending on the version, a Raspberry Pi is much less powerful than a normal computer. This means that our application might need some specific tweaking to ensure an acceptable running time.

3.4 Final hardware choices

Considering all the options we described in the previous section, we decided to design the architecture of Genie with multiple Blinkstick devices controlled by a Raspberry Pi Zero. This choices were due to multiple different factors.

3.4.1 Blinkstick considerations

As stated before the maximum power output is of 2W when all the LEDs on the device are on with maximum brightness, with this configuration a single Blinkstick device will draw up to 500mA [3].

This information is crucial to know in advance to choose the appropriate hardware configuration and power supply for our project to figure out how many devices we can connect simultaneously, their total power draw and most importantly if a Raspberry Pi would be able to output enough power for all of them.

With this information on the Blinkstick LEDs we started to research the Raspberry Pi version that would better fit our purposes.

3.4.2 Raspberry Pi considerations

The Raspberry Pi is offered in eleven different versions [4].

Although very similar between them, each of these versions offers some slight variations. In particular the most evident concern the CPU speed of the device, its available RAM and its power consumption. The specifications for all the versions are as follows, note that only nine of the eleven versions are listed since older versions are currently discontinued [4].

TABLE 3.2: Raspberry Pi specs [5]

Product	SoC	Speed	RAM	USB Ports	Ethernet	Wireless	Bluetooth
Raspberry Pi Model A+	BCM2835	700 MHz	512 MB	1	No	No	No
Raspberry Pi Model B+	BCM2835	700 MHz	512 MB	4	100Base-T	No	No
Raspberry Pi 2 Model B	BCM2836/7	900 MHz	1024 MB	4	100Base-T	No	No
Raspberry Pi 3 Model B	BCM2837A0/B0	1200 MHz	1024 MB	4	100Base-T	802.11n	4.1
Raspberry Pi 3 Model B+	BCM2837B0	1400 MHz	1024 MB	4	100Base-T	802.11ac/n	4.2
Raspberry Pi Zero	BCM2835	1000 MHz	512 MB	1	No	No	No
Raspberry Pi Zero W	BCM2835	1000 MHz	512 MB	1	No	802.11n	4.1
Raspberry Pi Zero WH	BCM2835	1000 MHz	512 MB	1	No	802.11n	4.1

The main hardware difference between the various models of Raspberry Pis comes from the fact that all of them except the Zero have a quad-core processor while the Zero implements a single-core one.

In addition to this the various versions of Raspberry Pi Zero are distinguished by the fact that the Raspberry Pi Zero W has wireless connection capabilities while the Raspberry Pi Zero WH comes with pre-soldered header pins.

Another aspect that we needed to take into consideration when choosing the Raspberry Pi version that would suite our purpose best was the power amount that it was possible to draw simultaneously from the USB ports.

TABLE 3.3: Raspberry Pi power specs [5]

Product	**Recommended PSU current capacity**	**Maximum total USB peripheral current draw**	**Typical bare-board active current consumption**
Raspberry Pi Model A	700mA	500mA	200mA
Raspberry Pi Model B	1.2A	500mA	500mA
Raspberry Pi Model A+	700mA	500mA	180mA
Raspberry Pi Model B+	1.8A	600mA/1.2A (switchable)	330mA
Raspberry Pi 2 Model B	1.8A	600mA/1.2A (switchable)	350mA
Raspberry Pi 3 Model B	2.5A	1.2A	400mA
Raspberry Pi 3 Model A+	2.5A	Limited by PSU, board, and connector ratings only.	350mA
Raspberry Pi 3 Model B+	2.5A	1.2A	500mA
Raspberry Pi Zero W/WH	1.2A	Limited by PSU, board, and connector ratings only.	150mA
Raspberry Pi Zero	1.2A	Limited by PSU, board, and connector ratings only.	100mA

As it is clear to see from Table 3.3 most versions have strict limits on the maximum total USB peripheral output, when considering that a single LED will draw up to 500mA if we want to have multiple ones connected at the same time the only reasonable choices are either a Raspberry Pi Zero or a Raspberry Pi 3 Model A+ since these are the only models that do not put a hard cap on the power output.

Before choosing one of these versions we need to have a clear plan about how many LEDs we want to connect simultaneously and what power supplies are available on the market, bearing in mind that all Raspberry Pis are powered by a 5V micro USB input [4].

Since the initial phases of the project one of our envisioned applications was a device that could visualize the Google Calendar used by the REVEAL group to keep track of which researchers are in office, for this we knew that we needed to have at least ten LED lights.

Since none of the Raspberry Pis offer ten USB ports we were forced to connect them to one

or more USB hubs, if possible we also wanted to have a single power cable to keep a more streamlined look so we opted to prioritize solutions that would have not required the use of powered USB hubs.

3.4.3 Power supply

By referring to a table present on the Raspberry Pi official website FAQ page we calculated the rough amount of Amps that a 5V power supply would have needed to output to achieve this goal, we referred to the maximum power draw under stress to be sure that we would have enough power for all situations.

TABLE 3.4: Raspberry Pi power draw [5]

		Pi1 (B+)	Pi2 B	Pi3 B (amps)	Zero (amps)
Boot	Max	0.26	0.40	0.75	0.20
	Avg	0.22	0.22	0.35	0.15
Idle	Avg	0.20	0.22	0.30	0.10
Video playback (H.264)	Max	0.30	0.36	0.55	0.23
	Avg	0.22	0.28	0.33	0.16
Stress	Max	0.35	0.82	1.34	0.35
	Avg	0.32	0.75	0.85	0.23

From Table 3.4 we can see that the Raspberry Pi Zero under stress draws a maximum of 350mA, summing this with the 500mA max power draw of our LEDs we get:

$$350 + (500 * 10) = 5350mA = 5.3A$$

This meant that a 5V power supply would need to output at least 5.5A to allow all the LEDs to be on at full brightness while connected to a Raspberry Pi Zero under load.

By Ohm's law we know that:

$$I = \frac{V}{R}$$

Where I is the the current through the conductor expressed in Amperes, V is the Voltage and R is the Resistance expressed in Ohms of said conductor. Since for our need the value of I is higher than the value of V it means that we would need a power supply with a resistance of less than one Ω.

Having this low of a resistance is quite hard to achieve in a power supply [12] and there are not many available on the market, hence we needed to find a better solution.

We know that RGB LEDs are composed by three different diodes, a Red, a Green and a Blue one that mix together to achieve a full range of colors [11].

The maximum power draw is calculated when all the LEDs are on and the resulting color is white, this means that all three channels are active at full brightness, hence we can reduce the power draw by using less channels. For our application we need only two colors, one to

represent when a researcher is in office and one to signal that he is not. By using green for the first case and red for the latter we can reduce the maximum power draw of each LED to:

$$\frac{500}{3} \approx 170mA$$

This means that our total power draw with ten LEDs connected at maximum brightness while using only one color channel each becomes:

$$350 + (170 * 10) = 2050mA = 2.05A$$

This is a much easier value to achieve with a 5V power supply, in fact we can even consider the recommended PSU power capacity for the Raspberry Pi Zero in our calculations instead of the maximum power draw under stress to get:

$$1200 + (170 * 10) = 2900mA = 2.9A$$

With this we know that a Raspberry Pi Zero with a 5V 3A power supply at the least will suffice for our needs.

In fact, we decided to go for a Raspberry Pi Zero W for six main reasons that mate well with our project goals:

- it is cheap to buy and maintain;
- is really small so we can easily design the hardware around it;
- we will not do anything computation heavy, so its CPU is perfectly enough for us;
- its energy consumption is very low, just perfect for us;
- it has WiFi, so we can easily connect to it remotely and it can easily get its input data from any data sources;
- we can easily use it with Raspbian which gives us a lot of freedom. It does not restrict us in the programming languages, so we can implement our toolset in language such as Python/Java/C/C++, and this is important because we have to work with the libraries of the LEDs, which can provide APIs in various languages.

Chapter 4

Genie Development

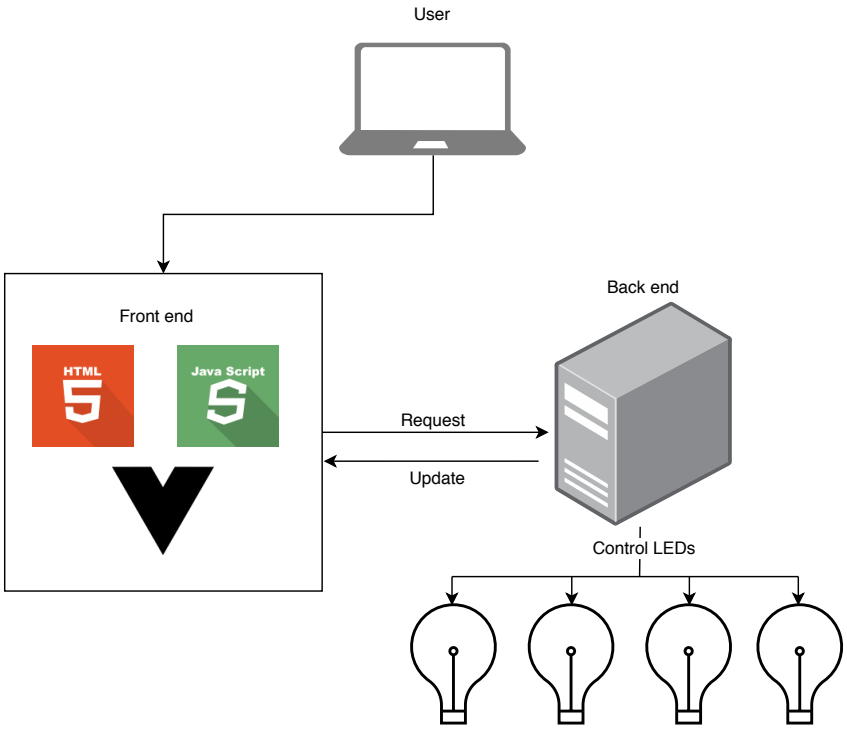

FIGURE 4.1: Architecture diagram of the application

In this chapter we will discuss the development process and the functionalities implemented in Genie.

4.1 Overview

For our application we decided to create a web-server using REST APIs [16] to communicate with the LEDs, this choice was dictated by the fact that we wanted to have a headless device that could be controlled even without having direct physical access to it.

For this we needed to create both a back end and a front end for our application that would be easy to expand upon, since we decided to work with Blinkstick LEDs and a Raspberry Pi

Zero the obvious choice regarding the language to use to create the back end was Python since it is natively supported by both devices.

4.2 Supporting libraries

To control the Blinkstick LEDs a Python, among other languages, library is provided by the manufacturer.

This library provides only the basic functionalities to control a Blinkstick device, hence we had to develop a set of functions on the top of it that we need for our application execution.

4.2.1 led_utils

In particular by using the default library only the first LED light of a device will be lit, this defeats the purpose of having multiple lights on a single device but it is a limitation to make the library compatible with all different models of Blinkstick available that may have different numbers of LEDs mounted on them.

Since we plan on using only devices with eight LEDs we created some functions that will light up all eight of them while implementing some checks to make sure the program will not crash even if a device with a different number of LEDs is connected instead.

Another safety net that we implemented in our library was a simple "retry if fail" system, due to the fact that we plan on having an unspecified number of devices connected at the same time we need to make sure that there are not any conflicts between them when sending the commands through the USB port.

In particular if multiple devices are connected to the same USB port, such as having a USB hub, and we send a command simultaneously to more than one of them we can not guarantee that both will be executed.

For this reason we decided to have the library retry the command, if failed, up to three times before returning an error. This is repeated for all LEDs present on a single device which means that in our case the worst case scenario execution will need to execute a command twenty-four times instead of only eight, three times for every LED instead of once.

We chose this number after some trial and error phases, we noticed that usually retrying only once was enough for the device to receive the correct information and very rarely we had to retry two times, usually when twelve devices were connected simultaneously. In our testing we never encountered a situation where we needed to retry the connection three times so we set this as the upper bound for our library but it may need to be increased if the number of devices increases significantly.

This number should nonetheless be kept as low as possible to avoid having the server stuck for a long time should the connection to the device be lost, e.g. by accidental disconnection of the USB cable or malfunction in the device.

Besides this safety feature, we implemented a number of basic functionalities to allow a user to turn all the LEDs on a single device simultaneously.

The official Python library documentation provided by the manufacturer to control the Blinksticks does not indicate that it is possible to individually set the color of a single LED independently from the others on the same device using the standard *set_color* function that it is used to light the first LED.

The official documentation only states that this function quote:

> *"Set the color to the device as RGB"* [6]

And takes five optional parameters that allows the user to select which color the LED should have:

- **red** (int) - Red color intensity 0 is off, 255 is full red intensity
- **green** (int) - Green color intensity 0 is off, 255 is full green intensity
- **blue** (int) - Blue color intensity 0 is off, 255 is full blue intensity
- **name** (str) - Use CSS color name as defined here: http://www.w3.org/TR/css3-color/
- **hex** (str) - Specify color using hexadecimal color value e.g. '#FF3366'

By analyzing the function signature, however, we can see that there is another parameter that can be passed to this function. Namely optional parameter *index* that by default is set to zero. Delving deeper in the source code of the function we can find that the index is used as follows:

```
1        if index == 0 and channel == 0:
2            control_string = bytes(bytearray([0, r, g, b]))
3            report_id = 0x0001
4        else:
5            control_string = bytes(bytearray([5, channel, index, r, g, b]))
6            report_id = 0x0005
7         ...
8         self._usb_ctrl_transfer(0x20, 0x9, report_id, 0, control_string)
```

From this we can see that the index is actually used to choose which LED on the device has to be responsible for the command just received.

Hence by changing the index we can control the corresponding LED on the device, this means that to turn on all the LEDs at once the only thing necessary is to create a function that simply calls this method in a loop changing the value of the index at each iteration.

With this knowledge we can implement some simple functions to control the device like turning off all the LEDs on it by setting the color to black (both Red, Green and Blue set to zero) or create simple animations like a wave by turning each LED on or off in a specific order.

These additional functions are really simple to implement given the limited number of LED lights on a single device and the simplicity to control them, as an example the code to create a wave is comprised of only eight lines

```
def wave(bstick, red=0, green=0, blue=0, rep=1, interval=0.1, name=None, hex=None):
    for rep in range(0, rep):
        for index in range(0, _led_number):
            bstick.set_color(0, index, red, green, blue, name, hex)
            if index > 0:
                bstick.set_color(0, index − 1, 0, 0, 0)
            time.sleep(interval)
    turn_off_all_leds(bstick)
```

This piece of code also allows the wave to be repeated any number of times with a single function call and the interval parameter allows to control the speed at which the wave will move.

It is also possible to create a function that will check if a certain device is completely off or has some LEDs turned on

```
def leds_off(bstick):
    time.sleep(0.02)
    for index in range(0, _led_number):
        if str(bstick.get_color(index)) != '[0, 0, 0]':
            return False
    return True
```

As it is clear to see, even if the basic library provided with the devices only provides the basic functionalities to control the LEDs they are sufficient to create a wide range of additional functions.

4.2.2 time_utils

The *time_utils* library is very simple and consists of a class that is responsible for keeping track of the time. All LEDs that need to read the current time should be using this library instead of accessing it directly, this is to standardize the way the time is checked between different LEDs connected to the server and make sure the value received is consistent between them.

An additional benefit of having a way to check the time separate from the single job implementation is the possibility of simulating the passage of time for testing or simulation purposes. In particular when the server is started it is possible to add in integer argument that will indicate the speed at which the clock will move in seconds at one tenth of a second intervals, if the argument is omitted or is equal to zero the clock will move at real time speed.

There are some checks in place to make sure that the timestep value is meaningful and will not create problems in the calculation or the simulated time, in particular said value is limited to a number between zero and 8640. This means that it is possible to have the clock read the real time or go in steps ranging from 10 seconds to one day each time a real second passes.

4.3 Back end

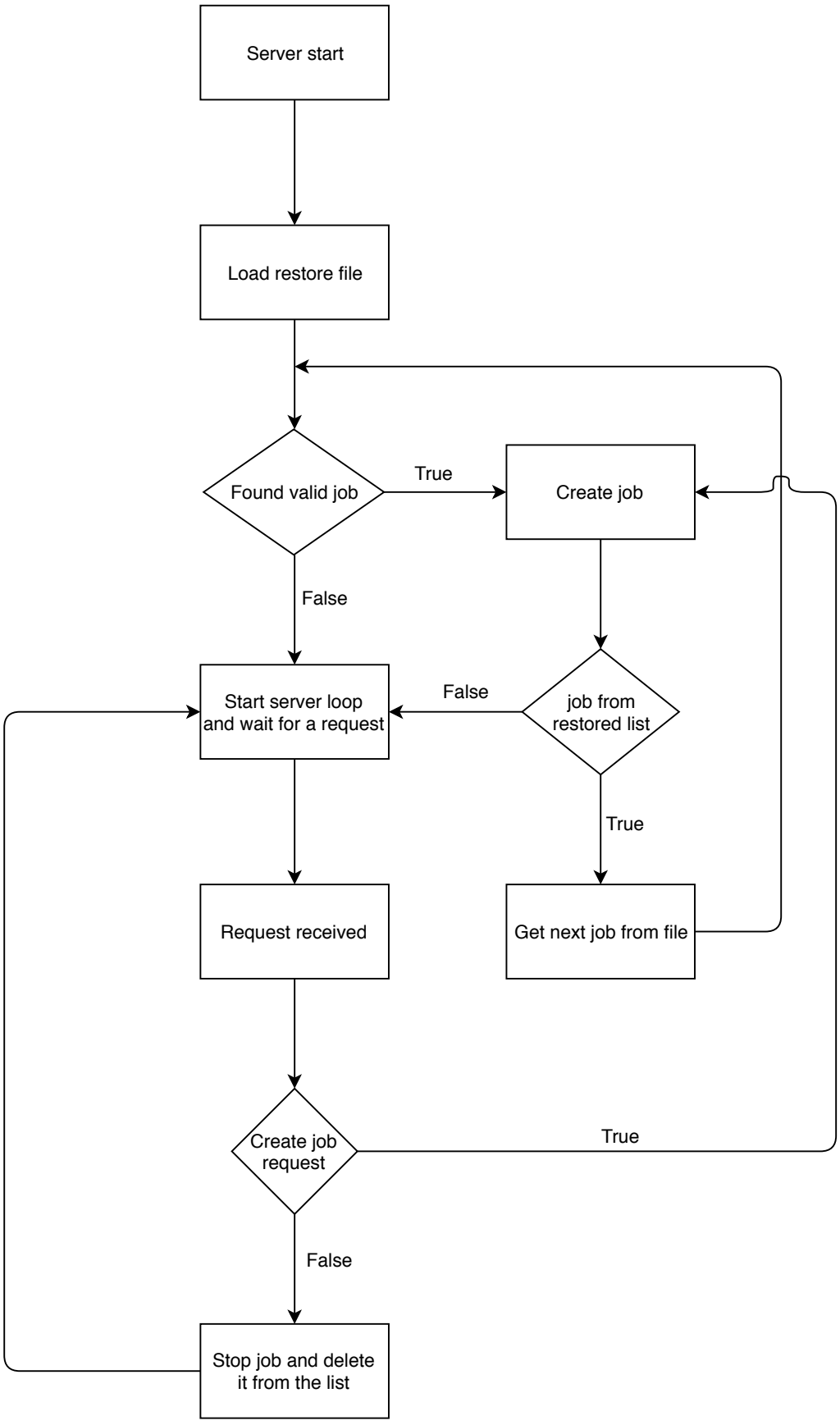

FIGURE 4.2: Genie's backend execution flow

To develop the back end of our application we chose to use Flask, a framework that makes building python apps much quicker by automating and simplifying some processes [19]. It particularly suits our application since it is made from the ground up to be simple and extensible, in addition to these features it is also relatively lightweight which pairs well with the limited resources of the platform we chose to host the service.

To control when an LED needs to change state we make use of Blinker [1]. Blinker is a library that allows different parts of code that are not connected to communicate using signals that are caught, by catching these different signals it is possible to execute different actions for every signal type that is caught.

The server is kept deliberately simple by having only a couple of routes available namely four POST and three GET routes used to create a job, connect an LED to a running job, stop a job that is running on the server, disconnect an LED from a running job retrieve the list of LEDs connected to the server and their current job, retrieve the list of running jobs and a route used to check if the server is up and running respectively.

The first GET route returns the information about the LEDs currently connected to the server as described in Table 4.1

TABLE 4.1: GET route response

Field name	**Content**
Serial	The unique serial number of the LED
Busy	Boolean that shows if the LED is executing a job or not
Name	The name of the job being executed or empty string

Every LED connected to the server is oblivious about the existence of other devices, this means that the various LEDs that are connected simultaneously to the server can not communicate with or influence each other.

For this reason we decided to create a new thread for every job running and device connected, when a command is sent to the server to connect an LED to a running job the serial of the device is saved in the object that contains said job and a very brief statement describing it for front end visualization purposes. This way we can easily ask a job for the LEDs connected to it and know the state of the system.

In addition, another reason behind this approach is the fact that if an LED color needs to be changed the only way to do so is to send a command to it with the new state that it should acquire. Not using a different thread for each device would have made this process far too complicated or would have required the server to become unresponsive to the user while updating the various devices statutes.

Each executing thread contains a while loop that runs endlessly until a function is called that sets the exit variable to True and allow the thread to complete, this is triggered by a POST request on a different route than the one used to start the job. The server is set up as to allow only one job to be executed by each LED maximum, to connect an LED to a different running job it first need to be disconnected from the previous one.

When a job is stopped the corresponding thread terminates and the LEDs connected to it are freed to signal that the LED is now available to take on a new one, all the light on the device

are then turned off to make sure none of them may remain accidentally set to a color.

4.3.1 Additional Job Customization

When a job is started some additional informations can be passed to it to customize it more, if possible. These informations are passed by using a text input when instantiating the job, it is then the job responsibility to check if what was passed makes sense and to handle it.

This can be useful since it allows to create a single job with multiple options that are executed only if a certain input, or combination of inputs, is written by the user when starting the job.

4.3.2 Job restore

The POST route used to start a new job is really simple and just relays the data it receives to a different function that is in charge of setting up the thread, connect it to the correct LED and start it. This is done to allow for a simpler way to restore the currently running jobs should the server stop working for any reason such as power outage or forceful shutdown.

When a new job is selected to be executed right before the corresponding thread is started all the information needed to create it are saved in a dictionary that is later saved in a file. Correspondingly, when a job is stopped its entry is removed from the dictionary on the file.

To avoid possible corruptions the updated list of currently running jobs is first saved in a temporary file and only when the save is completed the old list is overwritten by the new one, this ensures that even if a problem stops the server while the list is being written there will always be a version that preserves the integrity of the data.

By having a separate function dedicated to the creation and execution of the threads we can restore old running jobs by simply sending again all the data needed to it, this reduces code duplicity and makes the execution easier to understand.

The file containing the information to restore the previously running jobs is read each time the server starts, the only way to stop a job from being automatically started is to manually stop it before stopping and/or restarting the server.

Flask provides an annotation *@app.before_first_request* that allows to execute a function when the first request to the server is received and before it is executed.

The only problem with this approach is that if our application is rarely accessed a long time could pass between a server restart and the first request, if the server is shutdown for the night, as an example, we would want for it to resume its jobs as soon as it is restarted without having to manually do so otherwise the purpose of a restore functionality loses a lot of its utility.

For this reason we created the second GET route, the only purpose of this route is to receive a request to start the job restore process. For this purpose right before the server is turns on an additional thread containing a while loop is started, this thread automatically and periodically sends a request to that specific route and stops itself once it receives a response.

This allows us to trigger the *@app.before_first_request* event and the following restore process automatically once the server is started and before any other action can be executed on it.

4.3.3 Extensibility

The application is extensible with a simple plugin system, when creating a new job to be executed on the server the only requirement is to create a subclass of the `BasePlugin` class provided, implement the abstract function `job_definition` and place the file in the correct plugin folder. The base class manages the sending of the signals to the LEDs, the connection and disconnection of the various LEDs to the job and the shutting down of the job.

This allows the server to see the newly created plugin and execute it, the `job_definition` function takes as arguments the list of LEDs connected to the job and a number that indicates the number of seconds that will pass between one execution and the next one.

The `job_definition` is called once for every LED connected to the job, then after the time passed as argument to the function has passed the loop repeats. This allows the job to be run endlessly until the stop function is called.

Here we show a snippet of the `BasePlugin` class with the abstract method `job_definition` that would need to be overrode by the user in the subclass to create a new job:

```
1  def job_definition(self.data):
2      raise NotImplementedError("Please␣Implement␣this␣method")
3
4  def run(self, interval):
5      while self.keep_going:
6          if self.data != None:
7              for i in range(len(self.data)):
8                  job_definition(self.data[i])
9                  time.sleep(0.05)
10         time.sleep(interval)
11
12 def stop(self, caller, data):
13     if data == self.name:
14         self.keep_going = False
```

The overloaded function `job_definition` receives a tuple containing in the first position the serial ID of the current LED and in the second position the string that was provided by the user when connecting the LED to the Job as described in Section 4.4.

4.3.4 Genie API

When developing a new Job for `Genie` some functionalities are provided to control the LEDs more easily. The `BasePlugin` class already contains all the necessary functions that need to be called to send the signals to control the LEDs, when a new job needs to be created the user only needs to override the `job_definition` and call the already present functions `turn_led_on`, `turn_led_off` or `blink_led` that are responsible of sending the signals to control the LEDs.

In addition to the LED identifier it is possible to send some additional data with the signal like color, intensity and what single lights on the LED board need to blink.

Table 4.2 shows the available functions to control the LEDs and the parameters that can be passed to them while table 4.3 contains the description of each parameter.

TABLE 4.2: Genie API

Function name	Parameters						
turn_led_on	LED_serial	Color	R	G	B	Hex	Index
turn_led_off	LED_serial	Index					
blink_led	LED_serial	Color	R	G	B	Hex	Index

The `LED_serial` parameter is mandatory while all the others are optional, table 4.4 contains the default values for the optional parameters.

TABLE 4.3: Parameters Description

Parameter name	Description
LED_serial	The serial ID of the LED that needs to be controlled
Color	The name of the color the LED should have (E.G. "red")
R	The int Red value in RGB (0-255)
G	The int Green value in RGB (0-255)
B	The int Blue value in RGB (0-255)
Hex	The Hexadecimal value (E.G. "#ffffff")
Index	The int index of the light on the LED board (-1–7)

As stated in table 4.3 the index of the light on the board can be a number between -1 and 7, the values between 0 and 7 control the corresponding light while the value -1 is used to send the command to all lights with a single line.

If different color parameters are passed simultaneously they will be checked in this order:

$$Color - RGB - Hex$$

This means that, as an example, both the parameters for Color and Hex are present the latter will be discarded. The same applies if both RGB and Hex are present, the colors representing Black will be ignored and taken into consideration only if all three color parameters are set to it.

TABLE 4.4: Optional Parameters Default Values

Parameter name	Default value
Color	"black"
R	0
G	0
B	0
Hex	#000000
Index	-1

From table 4.4 it is evident that calling the function without passing any optional parameters will have the same result as calling `turn_led_off` with index equal to -1, this will turn all

the lights on the LED board off.

4.3.5 Genie Thread Architecture

As we stated previously, each LED is oblivious about the other connected devices and for this reason each one had a corresponding controlling Thread.

In addition to this if an LED has a blinking light an additional thread for each one is created.

This makes the thread architecture similar to a tree where the main server thread has as many child threads as there are jobs running, each job thread has as many child threads as there are LEDs connected to it and each LED has as many child threads as there are blinking lights on it.

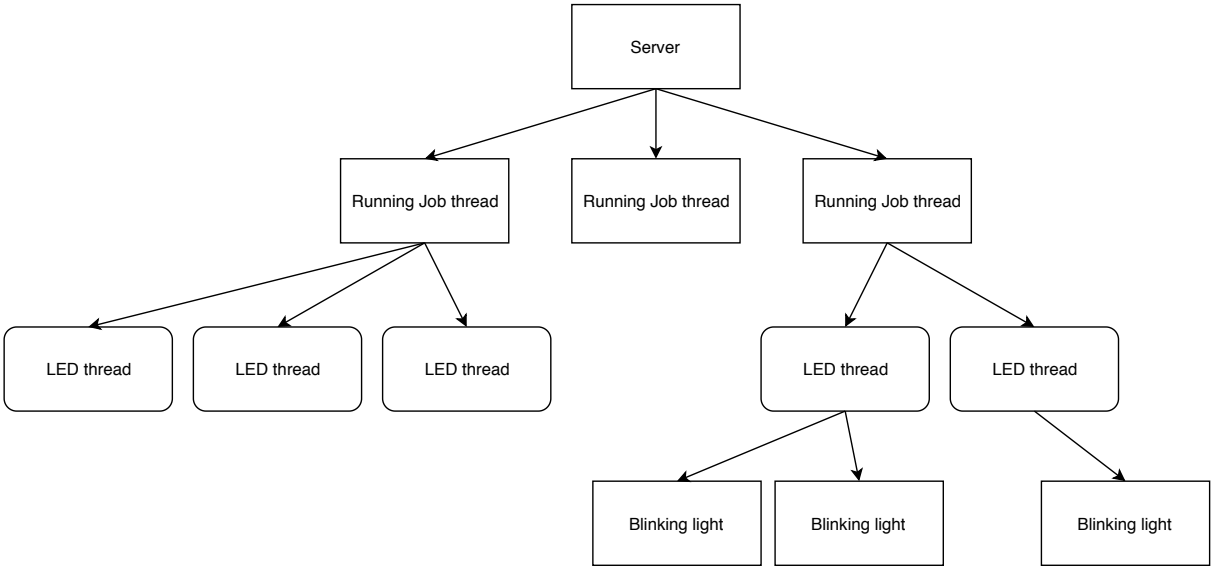

FIGURE 4.3: Thread architecture of Genie in an example state

In Figure 4.3 we can deduce that there are three jobs running, that the first one on the left has three LEDs connected to it, the one in the middle has no LEDs connected to it while the one on the left has two.

The main server thread only knows about the existence of the job threads, this means that when the signal to stop a job is sent it is the job's responsibility to stop all of the connected LEDs threads before finishing its own execution, is it then the single LED thread job responsibility to stop all its blinking threads.

This is really important because if the job thread is stopped without stopping its child threads any reference to those child threads will be lost but the threads will keep on working.

Once the job is halted every child thread that has not been stopped will still update its corresponding light and even starting a new job on the same device will not influence it.

This means that is a child thread is left dangling the only way to stop it is to manually kill it or shut down the server altogether.

4.4 Front end

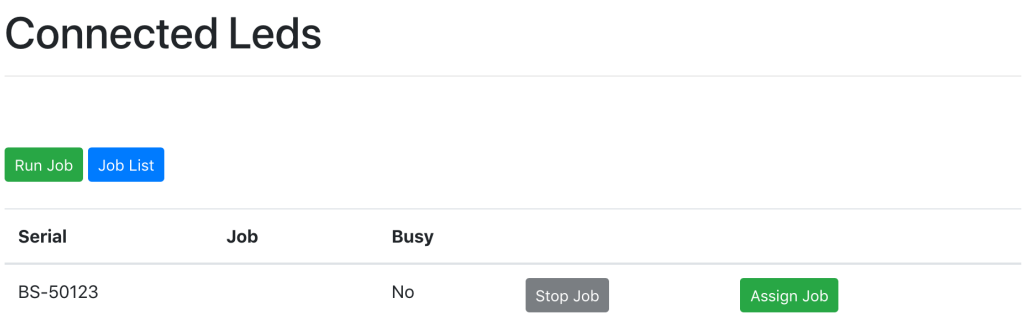

FIGURE 4.4: Genie's web interface with one connected LED

To develop the front end we chose to use Vue.js [7], a framework used to build user interfaces designed to be easily extensible.

In addition to this, it is also quite small when compared to its competitors without sacrificing execution speed or functionalities [8], just like Flask on the backend this makes Vue.js a very suitable option for our application.

The front end is structured as a single page application to reduce the amount of data that needs to be sent by the server while maintaining an easy to understand and navigate interface.

When a user connects to the front end, the page visible is composed by the list of LEDs connected to the server with their respective status and executing job name. If no LEDs are connected, the phrase "No LEDs found" is shown which lets the user know that the server is running correctly but no devices can be found.

This may be caused by either not having any devices attached to the server or by an error occurring while enumerating the USB devices connected to the server, if the latter a simple refresh of the page can trigger this operation to try and fix the problem.

If many devices are connected at the same time it is possible to simply scroll the list to find the correct one.

The order in which the LEDs are listed is not random but it is based on the order in which the server finds them, in practice this means that it is given by the order in which the devices have been connected to the various USB ports.

Due to this limitation the order will change if an LED is disconnected and subsequently reconnected to the server, the serial number will never change since it is unique and is stored on the device itself.

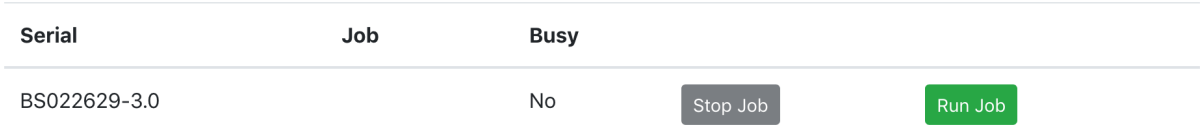

FIGURE 4.5: An example of free LED

To run a job it is necessary to press the "Run job" button, a component containing drop-down menu will appear to select appropriate one, once the job has been selected it is possible to customize some options via the available input and start it.

To assign a connected LED to a job a "Connect to Job" button is available, by pressing said button the list of running jobs is shown with buttons that allow to choose which job the LED should be connected to. By connecting the device to a job, the interface will automatically update to show the new status of the LED and allow the user to use the other devices connected to the server.

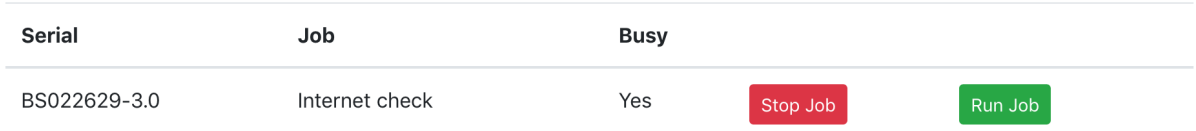

FIGURE 4.6: An example of busy LED

There are two different ways to disconnect an LED from a job and they differentiate themselves by one particular detail:

1. The first one is to press the "Disconnect" button, this will complete the thread responsible for the LED execution and turn of all the LEDs on the device.
2. The second one consists in simply dispatching a new LED connection request by pressing the "Connect to job" button again. This will disconnect the LED from the already running job but in addition to this, it will connect the device to the new job and add it to its list.

   This action is equal to disconnecting the LED from the job and connecting it to a new one but it is added to make it faster and easier to substitute the job an LED is connected to.

The device has an integrated 32 byte persistent memory which, in theory, would allow us to have a custom identifier for each LED to make it easier to recognize them. However, we found this memory rather unstable for our purpose (most of the time, the write would not complete successfully or would end up being corrupted). Hence, we had to abandon this feature and opted to rely on the serial number of the USB device instead.

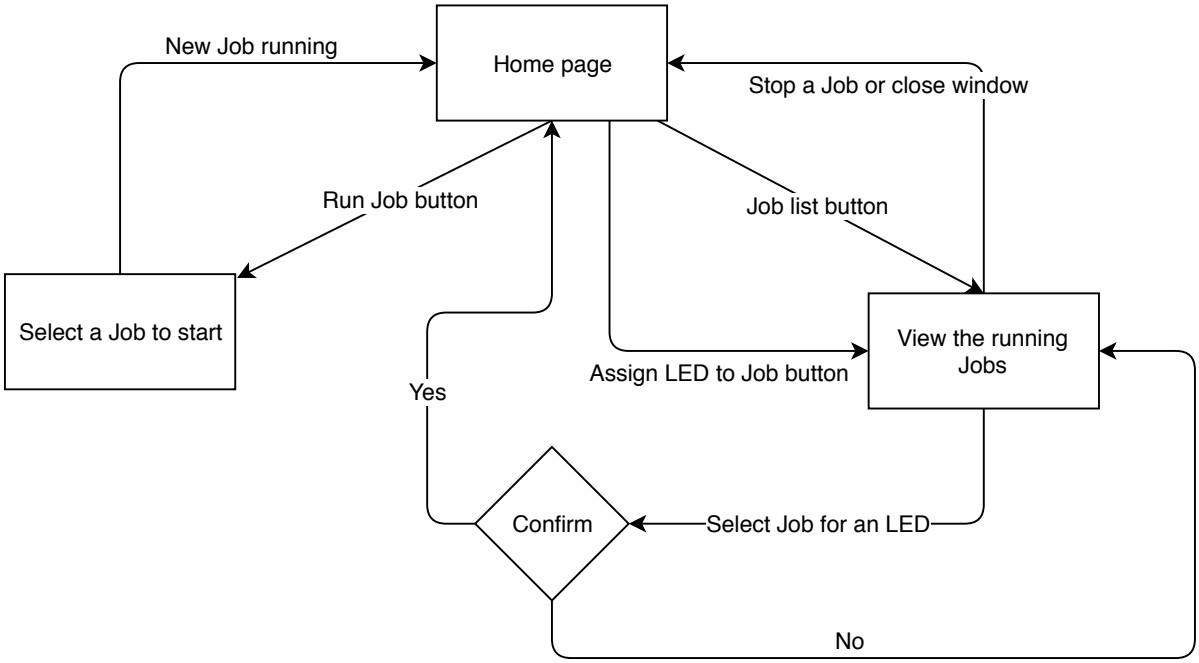

FIGURE 4.7: Genie's frontend diagram

Chapter 5

Example use cases

In this chapter we will expose some example use cases that we developed to show the potential of our approach by using data coming from various different sources.

5.1 Overview

All of the examples work on both the square and strip version of the device but some are more suited to one of them due to the lights positioning and diffuser distance from the source, in the description of the example it will be stated when said example has been developed to be used on a specific device. If this specification is omitted, no particular device was meant to be used.

5.2 Internet check

The first example use case that we created is a simple job that shows the user if the internet connection on the server is working or is down. This simple example requires many of our components, hence, it also serves as a first feasibility test of our approach. The job checks if the internet connection is available on the server by pinging a Google server once every second, if the server responds the lights on the LEDs turn Green otherwise a Red wave animation is shown to more effectively catch onlooker's eyes.

This is an example of data coming from the web, the response from google, due to an event triggered by the server, the ping.

FIGURE 5.1: Example of internet check job

Figure 5.1 shown a photo taken in a normally lit room with the device resting on white printer paper, this demonstrates the very high brightness of the LEDs.

5.3 Temperature tracker

This example was created to explore the use of third party APIs to control the LEDs color, In particular for this example we used OpenWeatherMap's APIs. OpenWeatherMap allows its users to get the current weather status for a location or a five day forecast split in three hours intervals for free [9], for this example we used the current weather in a specific location.

For this job the location can be either provided by the user or inferred as described in the following section.

With this example provide a way to visualize the temperature change during the day, for this purpose we show each minute the change in temperature from the previous one.

When the job is started, the light is set to Orange. Then after every minute check if the temperature has risen the light is set to Red. If it has decreased is set to Blue and if it has stayed the same is set to Orange.

This way it is possible to visually analize the change in temperature during the day and it is interesting see the light go through cyclical patterns such as the gradual lowering of the temperature in the evening or the rise in the morning.

5.3.1 Location inferring

Some jobs, such as the temperature tracker, require a location to be entered by the user.

When a job is selected it is possible to choose a city for which to get the current weather data or leave it blank, if the city selection is left blank the server will try to guess it based on its current IP.

With the IP it is possible to get a latitude and longitude approximation of the server's position using the *geocode* python library that can be included in the request to OpenWeatherMap's server to indicate the desired weather location.

If the name of a city is instead provided the server will try to find it's coordinates, this may result in the wrong city being detected if there exist multiple ones with the same name. To avoid this problem it is possible to specify the country in which the city is located in Alpha-2 code after the name of the city (e.g. Lugano,CH) this will ensure that the correct city, or one close to it if not directly available, is located.

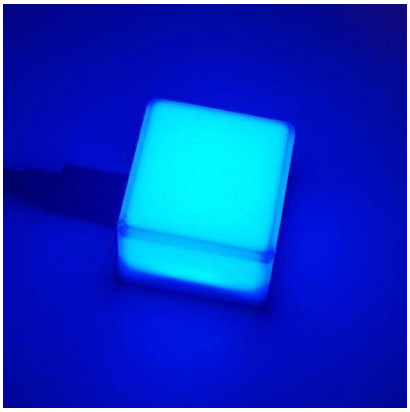

FIGURE 5.2: Example of temperature being lower than the previous minute

5.4 Weather forecast

The previous examples didn't take advantage of the beneficial feature of the Blinkstick devices that they have eight separate LED lights. All our jobs turned on all the lights at the same time with the same color except in the case of some simple animations like the wave implemented in the internet check example.

For this reason we created a job specifically designed to be run on the Strip, a visual weather forecast.

5.4.1 API change

Unfortunately for this application the OpenWeatherMap APIs that we used up until now do not provide enough information, the only forecasting option provided for free is a five day forecast divided in three hours interval.

This poses multiple problems to our implementation, having eight lights on a single device means that some lights will remain unused and the three hour interval makes it hard to get accurate data for the current and last day of the forecast.

This is due to the fact that the data for the current and last day is not complete but starts from the time we started the job and stops five days later without having the full day forecast.

For these reasons we had to change APIs in this example and in particular we chose the open DarkSky APIs [10].

These APIs are more strict in their usage limits compared to the OpenWeatherMap ones allowing only 1000 requests each day on a free plan but provide a forecast call.

This added functionality will return the next week weather data, starting from the day of the request, and include the mean temperature of each whole day so no matter what time the job is started we can get the temperature and weather values for the whole day.

Another nice feature of the DarkSky APIs is the inclusion of an apparent temperature value in addition to the normal one. The apparent temperature indicates the temperature that is felt by a human and is usually different from the real one due to various factors like wind and humidity [17].

5.4.2 Implementation

Taking advantage of the possibilities offered by the new APIs, we designed the ordering of the LEDs according to Figure 5.3.

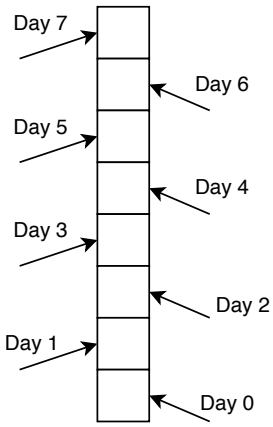

FIGURE 5.3: Weather forecast representation scheme

By using each light to represent a day we would have had a weekly forecast both easy and quick to understand. The days represented range from Day 0 (the current day) to day 7 (exactly one week from the current day).

Notice that, in practice, we could simply show the mean temperature of each day on the device. However, this would require to introduce some cutoffs for the different colors and it would not demonstrate well the usefulness of the application.

With this approach the device would have displayed interesting data while being used in months with a highly variable temperature but in winter and summer all the lights would most likely all have had the same color.

Hence, we implemented an alternative, second mode for the job called *"Normalized Week"*, with this mode the average weekly temperature is considered when selecting the color of each light instead of having some hard cutoffs.

This way the job retains its usefulness during the whole year, to choose the colors in this new mode we consider a change of 10% in the daily temperature.

If the temperature of a given day is within a 10% difference from the mean (either plus or minus) then that day will be represented by the color Orange, if the temperature increase

exceeds 10% the color will be Red and in the temperature decrease exceeds 10% the color will be Blue.

Moreover, we added a third functionality to our *led_utils* library to allow for a single LED on a device to blink independently from the others.

Due to the way the lights turn on and off we had to create a new thread for every single LED that we wanted to blink, this means that the Weather Forecast example has a structure used to keep track of the threads running on the device similar to the one present on the server so that all the threads can be stopped when the job is halted.

We used this new functionality to show the days in which rain or snow is predicted, with this example is now possible to get a complete forecast for the upcoming week with both temperature and precipitations warnings.

The device updates its status every 15 minutes with a new request to the DarkSky APIs to make sure the correct data is displayed regardless of changes on the forecast predictions.

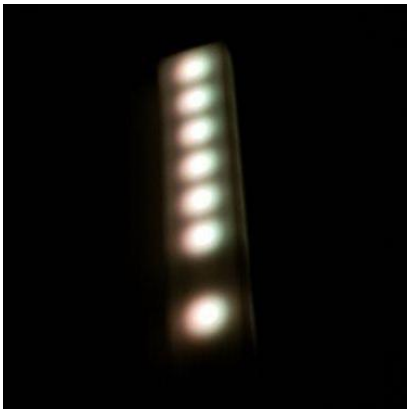

FIGURE 5.4: Example of weekly forecast

Unfortunately the very high brightness of the LEDs paired with the diffuser being really close to them makes it hard to take a good picture of this example.

Nonetheless in Figure 5.4 we can see that the temperature of the week is going to stay the same in all days and the second light representing tomorrow is blinking.

5.5 TCP listener

Here, we demonstrate an example that does not require the server to ask for the necessary data but rather receive it.

For this reason we built a job comprised of a TCP listener to allow the server to receive data and react to it.

When a user selects this as the job to be performed by the LED a port can be chosen to be dedicated to it, if the selection is left blank the default port is the number 1234.

In addition to the port it is possible to choose a value that will be compared with the one coming from the connection.

The server assumes that the port selected is open and can be used to receive the data, when the job is started the thread responsible for it waits for a connection on the selected port.

When a value arrives on the port the server reads it and turns on the LED accordingly, in particular if the value is equal to the one specified by the user the light will become Orange, if the value is higher the light will become Green and if the value is lower the light will become Red.

There is a limit to the speed at which the data can be received, the data can be read at minimum 0.02 seconds interval. If the data arriving at the server come in smaller intervals we found that the correct function of the LEDs is not guaranteed since sometimes the USB library used by the device will not communicate the data to the LEDs before the next command is required, there are some checks in place that will guarantee the colors to be represented correctly but there is no guarantee that the data arriving in smaller than 0.02 seconds intervals will not be lost.

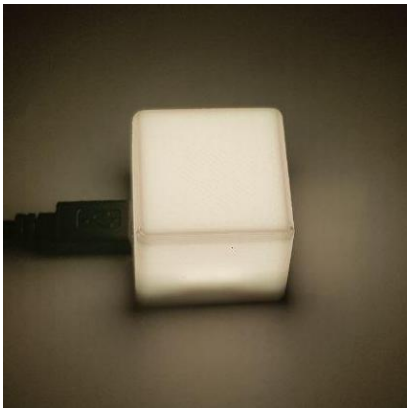

FIGURE 5.5: Example of tcp value

The example in Figure 5.5 shows a job running on the LED to check for an incoming connection providing the number 2. A user on a different machine is sending the number 2 to the server, since the number received is equal to the one we were checking the lights are colored Orange.

This job also works for any other comparable data type, in our example we used numbers for convenience.

5.6 Google Calendar

One of the applications we envisioned since the beginning of the project was constructing a device that could visualize the status of researchers from an existing Google calendar.

The first step to implement this example use case was to use the calendar's APIs to retrieve the list of events present on it, unfortunately Google does not use a secret key to handle requests to its APIs but instead relies on having an existing Google account and tokens saved on files residing on our own application server to handle the connection requirements.

This meant that we had to develop a system to use these files if present or generate them if missing.

When the job is run the first time if the API files are missing a browser window opens to let the user log into his google account and give the required permissions to the application, after all the permissions have been granted the required token files are saved in a dedicated folder on the server to be used for later requests to the API.

Google allows a great number of requests with its free API plan, this means that if we want to keep track of 10 researchers at the same time we could update their status every five seconds without running out of available requests. With this low of a refresh rate we can very quickly react and reflect changes in the dataset.

5.6.1 Data handling

The current method in which the calendar is used is not standardized and varies quite a bit depending on who is creating each event.

In general when a researcher has to leave the office he creates an event on the calendar with description "*name* away" and another event on a different day with description "*name* back" when returning.

Sometimes thought an event will not span multiple days but be limited to some hours of a single one, in these cases if the researcher needs to be in a given place he will usually write "*name* @ *place*" setting as start and end times the hours in which he will not be present.

Other times only the starting and ending hours of the commitment are set up and the description does not have a defined structure.

This makes the development of rules to decide if a researcher is in office or not quite complicated, for this reason we came up with a set of Heuristics that will serve most if not all of the current cases.

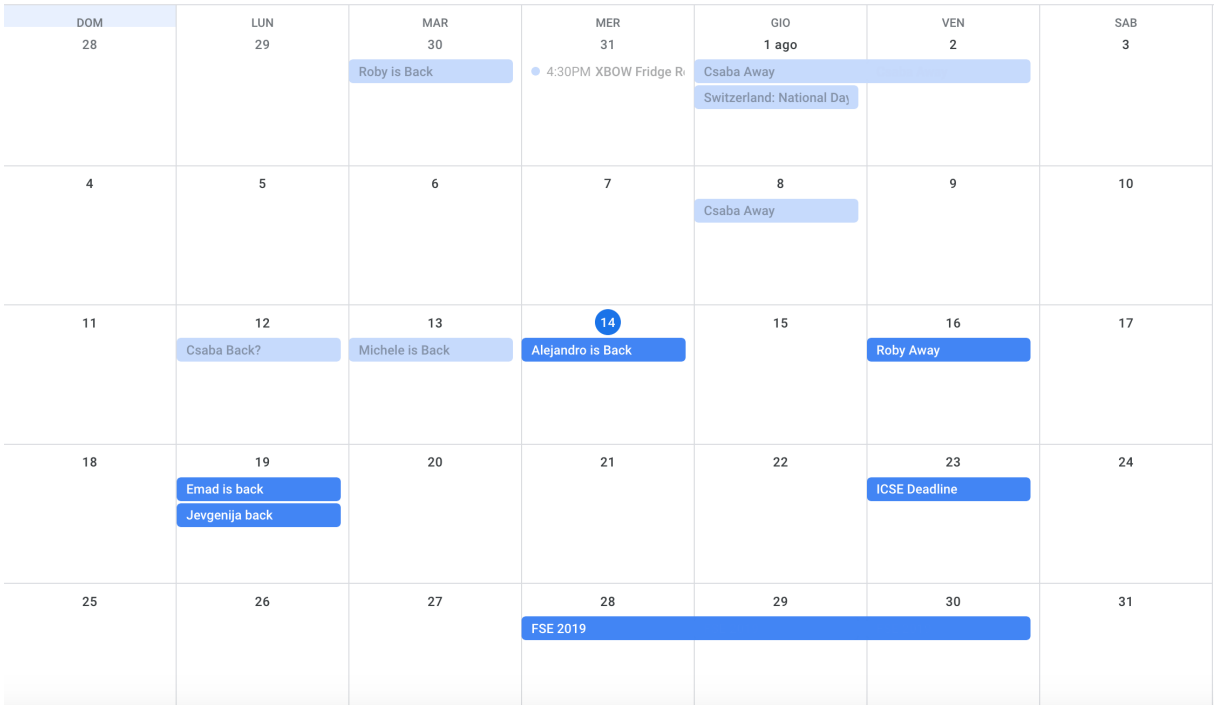

FIGURE 5.6: Example of the current Google Calendar visualization

5.6.2 Heuristics definitions

To understand and use the data coming from the request we developed some Heuristics. We cannot be sure that the person creating the event on the calendar is the one to whom the action is really related, for this reason to detect which person is the one actually connected to the event we parse the event's description in search of its name.

Similar to this we also search for some keywords to detect if the event is regarding a person being in office or not, in particular if we find the word "away" we assume that the person is not in office. Contrarily, if we find the word "back" we assume that said person has returned to the office and is therefore present.

In addition to these simple rules we decided that if an event has a start and an end time that are not both midnight, it means that the person in not in office between those hours. This rule was created to simplify the parsing of the event's description since there may be a great number of reasons that may force someone out of office, this rule encompasses all of them with a simple hour check.

It is also highly unlikely that someone will create an event to indicate that he is in office between two hours but will instead create one that indicates the hours when he will not be found.

For the same reason if an "@" sign is found while parsing the description we assume that the corresponding person is not in office but somewhere else in those hours.

As stated before we do not consider the case when an event start and end time is both set to midnight. This is because when a new event is created and no start and end times are specified

the default hours are set to the previous and current day midnight respectively.

This detail is not represented in the Google calendar user interface but it is present in the API calls, this means that if a user creates an event where he states that he is leaving without manually setting the times the event will last only one day.

As a final rule we assume that if no event is found in the past two months for a given researcher then he is currently in office.

5.6.3 Implementation

Once these heuristics have been defined the implementation of the example is quite straight forward, when the job is started the user can select one or multiple names that a device should keep track of.

This is useful if a researcher is not always referred to with the same name, by having multiple different names available we can check all the synonyms with the same job.

While the job is running each LED makes a request to the Google API server every five seconds to receive the list of events for the given calendar, the events are then sorted in reverse chronological order and traversed one by one while searching each of the names selected by the user.

For every event we find we execute one of the following actions:

- If a name is found in the description of the event we then first check if the start and end time for the event are midnight, if they are not and the current time is between them we set the LED color to Red to signify that the researcher is not in office.
- If the start and end times of the event are not midnight but the current time is not between them we set the LED color to Green and assume that the event has passed so the research is back in office.
- If the start and end times for the event are midnight we check the rest of the description, if it contains a "back" we set the LED color to Green because we know the researcher is in office.
- If the start and end times for the event are midnight and the rest of the description contains an "away" we set the LED color to Red to signify the researcher being out of office.
- If we find an "@" symbol in the description and the current time is between the start and end times for the even we set the LED color to Red.
- If the description does not contain any of the names given as input by the user we discard the event and repeat the check for the next one.

When one of the names given as input is found in the description of an event we stop searching for them and exit the event checking loop, this means that we do not have to traverse the whole list but only up until the first event containing one of the selected names.

This saves quite some time but makes it so that having multiple names of different people tracked by the same device is not useful since only one of them will be tracked.

5.6.4 Physical device build

Our vision for the Google Calendar example was to integrate it into the REVEAL group logo, for this we started thinking about what would have been the best and most efficient way to build the device.

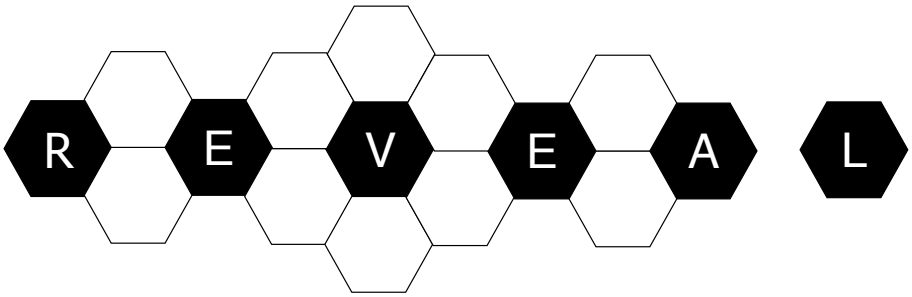

FIGURE 5.7: REVEAL logo

Taking the REVEAL logo as a base Figure 5.8 illustrates a diagram of where each component should be installed to allow for the best light dispersion while hiding the hardware.

The logo is formed by both white and black hexagons with the white ones on the outer sides, this means that the easiest and best solution is to put the LED lights behind the white hexagons and the Raspberry Pi with the various USB hubs behind the black ones where they will be hidden from sight.

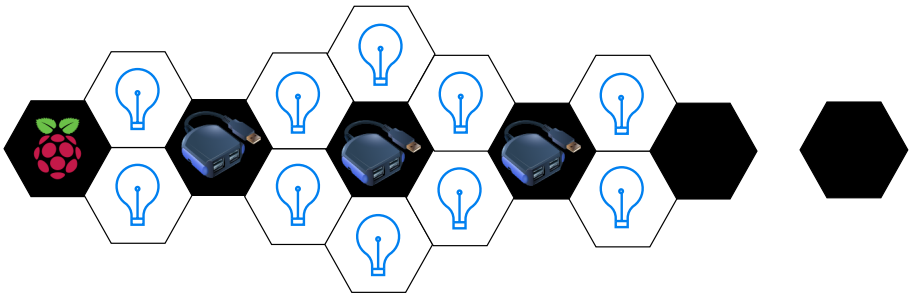

FIGURE 5.8: Devices position in the REVEAL logo

In addition to the Figure 5.8 diagram Figure 5.9 shows the connection diagram of the application. This is useful to keep track of how to connect the USB cables and their path.

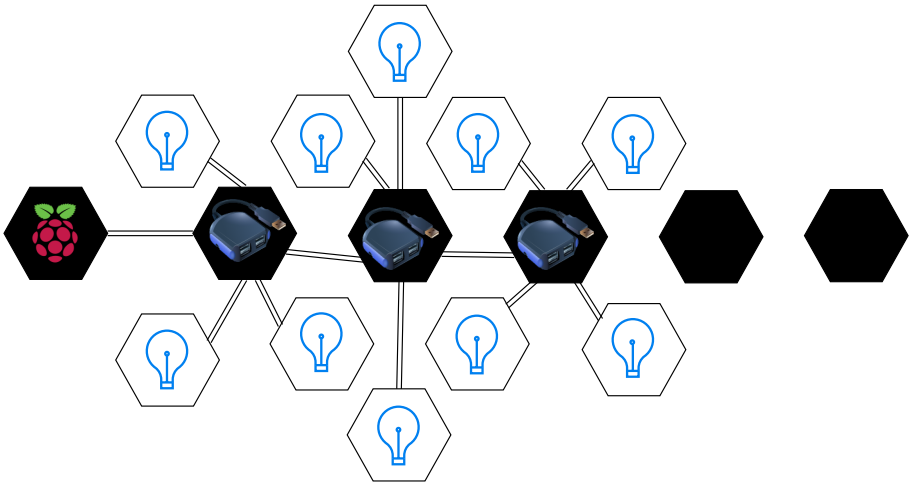

FIGURE 5.9: Calendar connection diagram

With this diagram in mind we just needed to order the remaining hardware and create the hexagons to host them.

The hexagons are 3D printed using the same file for all of them, this ensures that they would be as similar as possible, minus some small printing defects that can easily be removed once all have been completed.

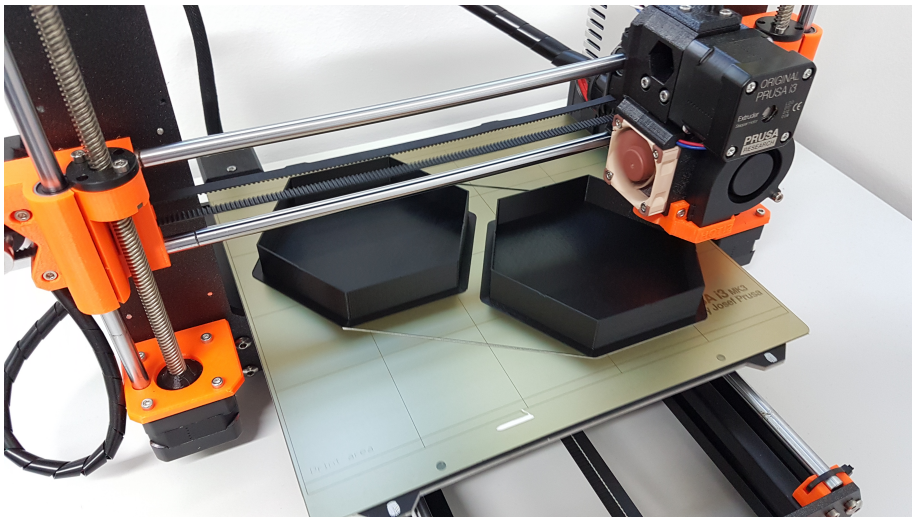

FIGURE 5.10: 3D printing the hexagons

With all the hexagons built to better hide the components and direct more light to the back surface the inside of the white hexagons is covered with some reflective material. One of the requirements of this application is for it to be easy to disassemble should the need arise, for this reason no permanent solutions are used to make it easier to disassemble should some component need to be changed or replaced.

FIGURE 5.11: The assembled Calendar device

After making sure that all the hardware fitted and the LED lights were secured in their position we set up some jobs to keep track of the researchers and test that all the connections were correct and working.

FIGURE 5.12: The finished Calendar device

This example shows how it is possible to move from an idea to a finished real world product that can react with the data coming from a server to interact with onlookers in a easy and understandable way without being too intrusive or disrupting.

5.7 Clock

The calendar application setup is well suited for multiple examples, in particular we created a clock example using the *time_utils* library to provide the LEDs with the correct and synchronized time.

Since each LED light does not know about the existence of the others to set up this use case it is necessary to input the total number of LEDs that will compose the clock, up to 12, and which position the current one occupies in it starting from one.

In our example the total number of lights in the clock is six and they are disposed as shown in Figure 5.13

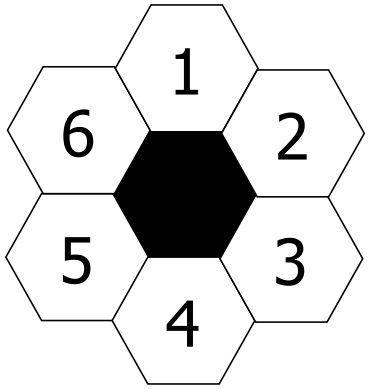

FIGURE 5.13: Clock lights disposition

Since all the lights on the clock share the same code there needs to be a mapping from the hours and minutes of the day to every single LED, in addition this mapping has to work for any number of lights (up to 12) that may compose the clock.

Starting from a 24 hour span we first take the current hour module 12 to get the hour value as a number between 0 and 11, we then divide 12 by the number of LEDs in our clock to get the number of hours that every light needs to represent and 60 divided by the same number for the minutes, after this can then define the mapping to each LED mapping as follows:

```
hour_led_to_light = (math.floor(current_hour / hours_per_led) + 1)
```

and

```
minute_led_to_light = math.ceil(current_minute / minutes_per_led)
```

This will tell us which LED on the clock we need to turn on for every hour and every minute.

In addition to this since we are trying to map 12 hours into six lights we decided to represent the even hours with a Green light and the odd hours with a Red light, the minutes are shown by a Blue light and when the hour and minute LED to be turned on is the same we show it by turning it Yellow.

Among all the examples we implemented this is probably the hardest to use in a real life environment. There would need to be far too many LED lights to represent the time in a meaningful way, nonetheless we think that this example shows the versatility of our concept.

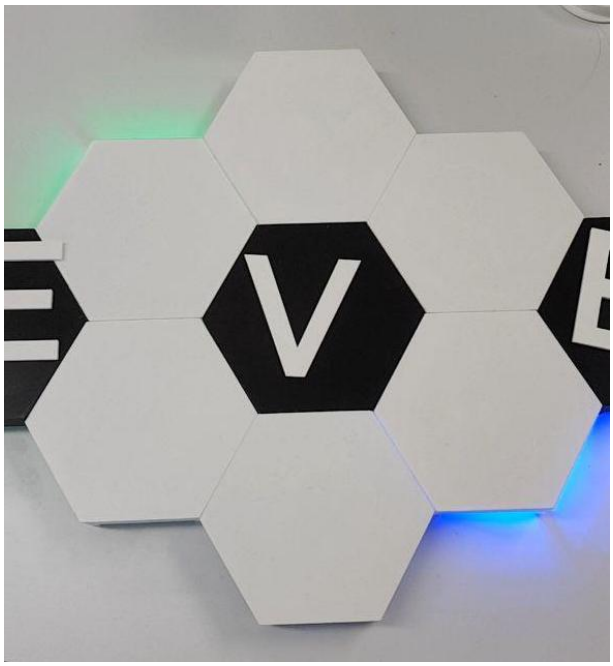

FIGURE 5.14: The built clock

In Figure 5.14 we can see that one of the LEDs is green while the other is blue, with out representation the blue LED indicates the minutes while the LED that represents the hours is green if the hour is even and red if odd.

In this example we can see that the picture was taken between 10:21 and 10:29 AM or PM.

Chapter 6

Conclusions and Future Work

In this chapter we will summarize our work and discuss possible future devolepments.

6.1 Summary

Our work in this document is meant to provide an easy and extensible way to represent data with LED lights, at its core Genie is a very simple application only comprised by a handful of functions but its power lies in its extensible architecture that allows a user to create new jobs in a quick way without having to worry about the underlying thread management aspect of the application.

Data visualization effectiveness is often hindered by the inherent complexity of the data it is based on. Most of the times, however, only some aspects of said data are really important for the user. We hope that our physical approach could prove useful to speed up the understanding of said data and reaction to changes as well as providing a more engaging and direct way to show it to onlookers.

The lightweight nature of Genie makes it suitable for deployment on almost any hardware configuration that provide a way to connect to the web.

As we stated in our introduction we wanted to create a system that could leverage the power of RGB LEDs to be used as devices to show complex aspects of the data rather than only simple on/off signals. We believe we accomplished this goal by creating a system that is both easy to expand to suit many different needs, easy to understand, robust and powerful.

With the creation of more and meaningful use cases it would be possible to create applications to show complex data in a non intrusive and non distracting but noticeable way.

6.2 Future Work

The easiest way to continue development with Genie is to create new jobs and study new options of data visualization using LED lights.

In particular it would be useful to add some monitoring functionality to the system so that the lights could be used to show the status of the machine they are connected to.

It would be very interesting to conduct a study on the user's response to this new way to visualize data, in particular if using LED lights to show important changes in the data reduces

interruptions or speeds up reaction times when compared to more widely used systems like notifications or dashboard visualizations.

A nice improvement would be adding support for multiple different LED types and manufacturers to not force a user to get a specific one like it is the case right now with Blinkstick.

Bibliography

[1] https://pythonhosted.org/blinker/, as checked on August 2019.

[2] https://blink1.thingm.com/about/, as checked on June 2019.

[3] https://www.blinkstick.com/products, as checked on June 2019.

[4] https://www.raspberrypi.org/documentation/faqs/, as checked on June 2019.

[5] https://www.raspberrypi.org/documentation/faqs/, as checked on June 2019.

[6] https://arvydas.github.io/blinkstick-python/, as checked on June 2019.

[7] https://vuejs.org, as checked on June 2019.

[8] https://vuejs.org/v2/guide/comparison.html, as checked on June 2019.

[9] https://openweathermap.org/, as checked on June 2019.

[10] https://darksky.net/dev, as checked on June 2019.

[11] M. Ali, S. Shivakumar, S. Bhavani, M. Bhargavi, and S. Altaf. *Multiple Colour Generation by using RGB LED*. IJSRD - International Journal for Scientific Research & Development| Vol. 5, Issue 01, 2017, 2017.

[12] R. Blanchard and P. E. Thibodeau. *The design of a high efficiency, low voltage power supply using MOSFET synchronous rectification and current mode control*. 1985 IEEE Power Electronics Specialists Conference, 24-28 June 1985.

[13] C. Chen. *Top 10 unsolved information visualization problems*. IEEE Computer Graphics and Applications Volume: 25 , Issue: 4.

[14] W. Gottesheim. *Challenges, Benefits and Best Practices of Performance Focused DevOps*. Proceedings of the 4th International Workshop on Large-Scale Testing Pages 3-3, 2015.

[15] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer. *What is DevOps?: A Systematic Mapping Study on Definitions and Practices*. Proceedings of XP2016 Article No. 12, 2016.

[16] L. Li, W. Chou, W. Zhou, and M. Luo. *Design Patterns and Extensibility of REST API for Networking Applications*. IEEE Transactions on Network and Service Management (Volume: 13 , Issue: 1 , March 2016), 2016.

[17] J. L. Nguyen, J. Schwartz, and D. W. Dockery. *The relationship between indoor and outdoor temperature, apparent temperature, relative humidity, and absolute humidity*. International Journal of Indoor environment and Health, Volume24, Issue1, pages 103-112, Feb. 2014.

[18] L. Pappas and L. Whitman. *Riding the Technology Wave: Effective Dashboard Data Visualization*. Human Interface and the Management of Information. Interacting with Information. Lecture Notes in Computer Science, vol 6771. Springer, Berlin, Heidelberg, 2011.

[19] L. P. S., A. Fankar, H. Nabeel, M. Jumal, and G. Murade. *Efficient way of web development using python and flask*. International Journal of Advanced Research in Computer Science, Mar. 2015.

[20] B. Ur, M. P. Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman. *Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes*. Proceeding CHI '16 Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems Pages 3227-3231, May 07 - 12, 2016.

[21] M. Züger, C. Corley, A. N. Meyer, B. Li, T. Fritz, D. Shepherd, V. Augustine, P. Francis, N. Kraft, and W. Snipes. *Reducing Interruptions at Work: A Large-Scale Field Study of FlowLight*. Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, 2017.