UNIVERSITY OF CALGARY

ENGO 500: GIS and Land Tenure #2

Final Report – First Submission

By: Jeremy Steward
Kathleen Ang
Ben Trodd
Harshini Nanduri
Alexandra Cummins
Supervisor: Dr. Steve Liang

DEPARTMENT OF GEOMATICS ENGINEERING

SCHULICH SCHOOL OF ENGINEERING

04/28/2014

ENGO 500 – Group GIS & Land Tenure 2
Jeremy Steward
Alexandra Cummins
Harshini Nanduri
Kathleen Ang
Ben Trodd

April 28, 2014

Dr. Steve Liang, Dr. William Teskey
Schulich School of Engineering
University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

To whom it may concern:

Attached you will find our first draft submission for the ENGO 500 final report, detailing the efforts and work of the GIS & Land Tenure 2 group to fulfill the ENGO 500 course requirements. The report details work done since the project's inception, towards the goal of creating an Internet of Things (IoT) application that interfaces with the Open Geospatial Consortium SensorThings API for IoT-based applications.

In large part this report is a collection of posts and documentation that can be found online through the project's Github repository (https://github.com/ThatGeoGuy/ENGO500). Furthermore, additional documentation can be found hosted through Github Pages at http://thatgeoguy.github.io/ENGO500. Should there be any questions regarding the content of this document, feel free to contact one of the team members at your convenience.

Sincerely,

Group GIS & Land Tenure 2

Table of Contents

Introduction....5
Purpose & Scope....5
Design Process....5
Project Proposal....5
Rationale behind project....5
Literature Review....6
Retail Analytics and Shelf Stock....6
Prototype / Hardware Specifications....7
Functional Requirements....7
Non-functional Requirements....8
Website / Software Specifications....8
Functional Requirements....8
Non-functional Requirements....8
Technical Deliverables and Progress Report....9
Use Cases....9
Sensors Used....11
Passive or Pseudo-Infrared Sensor (PIR Sensor)....11
Photo-Interrupter....11
Risks Identified....12
Prototype Design and Usage....13
Hardware Components....13
Breadboard....13
Pi Cobbler....14
PIR Motion Sensor....14
Wiring the Sensor....15
Coding in Python....15
Issues Encountered....16
Photo-interrupter....16
Wiring the Sensor....16
Coding in Python....17
LEDs for POSTing Observations....18
Putting It All Together....19
Software Components....20
Understanding and Using the OGC SensorThings Data Model....20
Using the OGC IoT SensorThings API....22
POST-ing information from the Raspberry Pi....24
Step 1: Create 'Thing'....24
Step 2: Obtain URI of Thing....25
Step 3: Obtain assigned Datastream IDs....25
Step 4: POST Observations....25
Allowing the Prototype to Run Without User Interaction....26
Running at boot time....26
Use Cases Implemented....27
1.1 – Update the Database With Formatted Database....27
1.2 – Format Data According to the OGC IoT Standard....27
1.3 – Identify Sensors....27
1.4 – Send Data to the Micro-Controller....28
Website / Interface Development....28

Webserver Design....28
Basics....28
Installing Pre-requisites....29
Running the Server....29
Server Configuration....30
Directory Structure....30
Root Folder (ENGO500-Webserver/)....30
Routes and Route Handlers (routes/)....31
Views (views/)....31
Public (public/)....31
Models (models/)....32
Configuration (config/)....32
Model Implementation....32
Use Cases....35
Use Cases Implemented....35
Use Cases Abandoned....35
Client / Front End Development....36
Store Layout Creator....36
Using the Layout Creator....36
Enabling Technologies....41
Store Layout Creator Architecture....41
System Overview....42
Data Structure....43
Technical Challenges With Layout Creator....43
Nested Accordions....43
Removing Shelves and Sections....44
Removing accordion elements....44
Removing SVG elements....45
Saving and loading editable attributes....45
D3.JS....46
Store Viewer....46
Enabling Technologies....46
Store Viewer Architecture....46
System Overview....47
Other Features....47
Future Work....49
Technical Challenges With Layout Viewer....49
Getting Observations with GET Requests....49
Getting Observations Asynchronously....50
Displaying Overall Statistics....50
Displaying Real-Time Observations....50
Use Cases....51
Store Layout Creator....51
Store Layout Viewer....51
Conclusion....52
Shortcomings of Project....52
Lessons Learned....52
Summary....52
Acknowledgments....53
References....54

Appendix A:.....56

List of Figures

Figure 1: Abstraction of the basic project model.....9
Figure 2: Use Case diagram for Shelf Prototype component of project from Figure 1 above.....10
Figure 3: Use Case diagram for Website component of project from Figure 1 above.....10
Figure 4: Example Photo of the PIR Sensor [16].....11
Figure 5: Example of Photo-interrupter from sparkfun.com.....12
Figure 6: Example of the breadboard we used for testing.....13
Figure 7: An example of the Pi Cobbler, an accessory for the Raspberry Pi.....14
Figure 8: Layout map of the gpio pins on the Raspberry Pi and Pi Cobbler.....14
Figure 9: Example of how wiring is set up for PIR Motion Sensor.....15
Figure 10: Schematic Diagram of Photo-interrupter.....17
Figure 11: Schematic of how we wired the LED with a resistor.....18
Figure 12: Photo of how we set up the LED on the breadboard.....19
Figure 13: Example set up of all the sensors and components together.....20
Figure 14: Data model defined by the OGC.....21
Figure 15: Adopted data model we used for our design.....22
Figure 16: Screenshot of what the hurl request should appear like.....23
Figure 17: Result from sending a POST request to the SensorThings API.....24
Figure 18: Screenshot of the navigation bar.....36
Figure 19: Initial page a user will see if they have an empty store layout (have yet to create one)...37
Figure 20: Depiction of when a shelf is added to the store layout.....38
Figure 21: Depiction of what happens as you add sections to your shelves.....39
Figure 22: A single section with editable attributes.....39
Figure 23: Complete Store Layout.....40
Figure 24: Linking Section to Data Model.....41
Figure 25: Screenshot of the layout creator in action.....42
Figure 26: Screenshot of system working with active observations being displayed.....47
Figure 27: Example of viewing historical data for your store.....48

Introduction

Purpose & Scope

The purpose of this document is to summarize and describe the totality of all the work done in the *ENGO 500: Project Design* course. The specific project detailed within this document is *GIS & Land Tenure #2: Developing an Open Source Internet of Things based Shelf System*, which particularly focuses on the development of a prototype smart-shelf and user-friendly web-based interface which communicates and interacts through the Open Geospatial Consortium's SensorThings API [1]. The shelf and interface together provide a rich, context-aware application for store managers and retailers to gain insights into the analytics of customer traffic and shelf stock within their store. With regards to the rest of the document, we will refer to the shelf system itself as a Location Aware Shelf System, or LASS for short.

This document in particular will summarize and discuss the previous document submissions throughout the project lifespan, and how they have affected design decisions of the final product. Furthermore, it will explain plainly the different components of the project, how they interact, and how they are used so as to provide a complete picture as to how such a product could be recreated if the project was started from scratch. Finally, a brief conclusion and evaluation of project goals is discussed. Previous reports submitted as part of the completion requirements for this course are likewise attached after Appendix A. There are no further appendices within this report, so any report attached there stands on its own merit.

Design Process

Project Proposal

When our group began the project, we were not entirely sure as to what we might choose to develop for the project. While we obviously had course requirements (as outlined for the ENGO 500 course guidelines), we were largely given free reign as to what we wished to develop, given the following constraints:

1. The project was to be an “Internet Of Things (IoT)” based project [2].
2. The intent of the project was to be developed in tandem with the Open Geospatial Consortium (OGC) SensorThings API [1], which was at the time in its infancy.
3. Since the purpose of our project was in a way a test drive of the OGC SensorThings API, we wanted our application to be open source, so that others could benefit from our work and learn from any mistakes or issues we may have encountered along the way.

Rationale behind project

When thinking about the Internet of Things, there are three major components that comprise an IoT-based system. These are

1. The “Thing” that is to be connected through the internet,
2. The interface with which to interact with the object or thing, and
3. Some system which allows both the thing and the user-facing interface to connect and *speak* with one another.

Looking at the market today, there are a few IoT-based applications that have already been successful. Some examples are the Philips Hue [3], or any of the WeMo Home Automation [4] products. Effectively, our *thing* that we wish to connect to the internet can be almost anything that

has some sensors connected to it; likewise, our interface can range from common websites, to full-fledged smartphone applications. Unfortunately, these various products do have some pitfalls, particularly due to the proprietary nature of the software they run on. This makes it difficult to directly acquire data from the sensors and use various products in tandem with one another (mostly due to user-facing software only being compatible with the proprietary information reported by the sensors). This can result in vendor lock-in for the user, but likewise means that users have to overcome the burden of investing entirely in one ecosystem when they want to buy IoT-based products.

Naturally, this was the rationale behind building the OGC SensorThings API, which advertises itself as:

> [A]n OGC candidate standard for providing an open and unified way to interconnect IoT devices, data, and applications over the Web.

For this reason, we were tasked with developing an application that used the OGC SensorThings API, because it served as an open standard with which anyone could develop interoperable IoT applications. That said, we set out to develop with a few key objectives:

1. Our “Thing” could be anything, but we needed to develop some software which allowed regular sensors to communicate with the SensorThings API.
2. We likewise needed some interface to make the application more than just sensors sending readings to a server. For this reason, we developed a website [5] that would allow users to interact with the system that we envisioned.

Some other minor considerations had taken place as well, which will be covered in more depth in later sections. If you want to read the original project proposal [6], see Appendix A.

Literature Review

During the project proposal, there was little talk as to what we would do to fulfill our objectives for our final project. This was mostly due to the amount of freedom we had in picking a topic; we wanted to develop something that not only seemed useful, but was likewise feasible to accomplish within the time constraints of the project.

We had several ideas from the beginning that might have benefited from being incorporated into the internet of things. Some examples include a system to measure the capacity of parking lots around the city, a system to measure and report soil moisture throughout fields or areas, and even a system that could detect and report when snow removal would be required in residential areas (Calgary has had some nasty storms of late). Ultimately, we decided to go with developing a shelving system that would be capable of measuring if a shelf is stocked or empty, as well as being capable of tracking the location of customers within the store.

The primary purpose of our literature review was to evaluate previous attempts at solving these problems, as well as attempt to evaluate some of the different equipment or methods we would need to employ in order to make this project successful.

Retail Analytics and Shelf Stock

Overall, we found that there is an incredible amount of work and research put into this topic already. Previous implementations have touted the use of RFID [7]tags in order to map movements and shelf stock. There are even several [8] examples [9] where such systems have been implemented and tested. While the efficacy of these solutions is not debated, RFID tags have shown some particular shortcomings, particularly:

1. RFID tags can potentially become very noisy, especially if there are a lot of them in

a small area.
2. Direct information cannot be obtained from RFID tags. Unlike sensors, RFID tags always act like on/off switches in that they're either connected or not.
3. RFID tags require advanced techniques to track customer motion throughout the store, and are typically not as scalable, since you cannot add more sensors or information into RFID systems as easily as in full sensor networks (such as the OGC SensorThings API).

Ultimately, we chose to avoid RFID-based solutions for the purpose of our project. Partially because of what's listed above, but likewise because we wouldn't be able to make an IoT-based project out of RFID tags. Other retail analytics solutions involve the use of closed circuit television (CCTV) footage to track customers, or require customer tracking based off of their smartphone's bluetooth / MAC address. These solutions are considerably harder to implement, and much more costly. In the case of CCTV networks, many stores have their own proprietary solution (from various vendors, many being local vendors) and this makes it very hard to integrate the CCTV footage into other projects. Since the purpose of our project is to help proliferate the open standard, these networks pose a much higher barrier to entry to get users on board with what we developed.

Prototype / Hardware Specifications

With all that being said, we decided to create a sensor-based solution to provide shelf-stock and customer location information. Thus, we needed to determine what kind of sensors we wanted to use, and how we were going to link all of the sensors together. To manage information throughput to/from the sensors, we ultimately decided on using a Raspberry Pi [10]. While we debated on some other solutions, such as the Arduino Uno [11] or the Netduino [12], we ultimately decided on the Raspberry Pi for the following reasons:

- It was simple: we could set it up with a full GNU/Linux operating system and start developing without requiring any special libraries or frameworks
- We could develop in Python rather than C or C#, which would ease the learning curve and development process for the team
- Our project supervisor had some available.

Unfortunately, since very little research could be found with regards to solving this problem using sensors, we did not come to any conclusion at the time regarding which sensors we would use, or how we would coordinate the sensors. We did, however, generate the following functional and non-functional requirements for the project:

Functional Requirements

1. The proposed shelf system should be able to tell when a product is no longer faced / in stock.
2. The proposed shelf system should be able to sense if a customer is standing in front of one of its sections.
3. The shelf should (as a final product) be sized similarly to existing store shelves, and should not obstruct or displace current shelf systems.
4. The final device should not break down if internet access is lost, but otherwise should expect that a reliable internet connection is in place in order to perform normally.

Non-functional Requirements

1. The shelf system should conform to the Open Geospatial Consortium SensorThings API standard.
2. As a corollary to *Functional Specification #4* above, a stable internet connection should be made available so that the shelf can operate and communicate through the SensorThings API.
3. Power must be made available to the shelf. While a final "ready-made" product was never developed, we expected the power draw to be similar to that of a Raspberry Pi (700mA / 5V).

Website / Software Specifications

As mentioned in the previous section for the project proposal, an interface of some kind is needed in order for users of the system to interact and view the information that our sensors are sending to the IoT service. While our group had the option of developing a smartphone application, we felt that the necessary overhead of learning how to create an application would be too much given the scope of all the other components of the project. Combining that with the fact that some members of our group had some previous experience building and developing websites, it was decided that the simplest way to provide an interface would be through a website that users could sign in and connect to. We came up with the following requirements that our website should fulfill:

Functional Requirements

1. The ability to display shelf stock / facing measurements in real time, so that users could figure out where to restock things.
2. The ability to track customers traveling through aisles within the store.
3. The ability for the website to generate some basic analytics based on data from the sensors.

Non-functional Requirements

1. The system should be painless to introduce, and should require no more than signing up for a service, similar to signing up for something such as GitHub or Twitter.
2. The system should provide some means of security so that unauthorized access to the system is prevented or stopped.
3. The system should not fail or cease to work if there is no power in the store or if the shelf system itself is prevented from working normally.
4. The website should be interoperable with the OGC SensorThings API, as was the case for the Prototype / Hardware component.

We also mentioned requirements regarding the ease-of-use and testing requirements for the final system, which are not explicitly listed here since they are more concerned with the final product rather than the development process itself.

From this point, we continued to plan a development methodology for both branches of work (the Prototype of the shelf itself, as well as the Website development). This work leads us to the next step in our overall Development Process. As before, you can read our full literature review [13] in Appendix A, and see a more in-depth look into how we approached it at that time.

Technical Deliverables and Progress Report

Unlike previous milestones throughout our project, the technical deliverables and progress reports

provided more of a progress summary of work done up until the point in time that it was submitted. The majority of the first four months of development was dedicated to planning and researching the methods required to make the LASS project come to fruition. In particular, with the definition of Use Cases and potential risks, we set out the framework which helped direct how we intended to develop the rest of the project. The two months after focused largely on implementing technical details which are described further into the report.

Use Cases

With regards to engineering and software development, use cases are a way of defining the input and output relationships between actors and elements within a system. Typically, use-cases are written in the form of tables, with one table per use case. However, as the number of use-cases for our project is greater than just a few, we felt that it would be best to leave the full use-case descriptions in the original technical deliverables report [14], as opposed to copying them here. This is mostly because the quantity of space it would take to present them would crowd out the rest of the discussion for this step in the development process. Alternatively, if you don't want to read the full report, but would still like to view the fleshed-out use-cases, you can always check the original wiki page located on GitHub [15].

By this stage of the project the group had split into two teams, in order to tackle the two major components of the project: developing the sensor prototype, and developing the corresponding website. Working in separate parts, our group came up with two sets of use cases that we would define so as to interact with the overall model of the proposed system, shown in Figure 1 below:

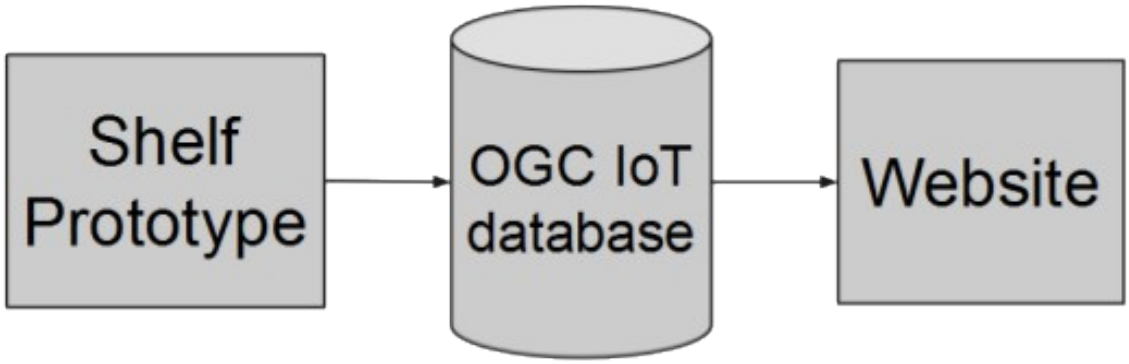

Figure 1: Abstraction of the basic project model

From this, we had the prototype use-cases focus on the following figure (Figure 2), which would ultimately upload data to the OGC SensorThings API (also known as OGC IoT database above).

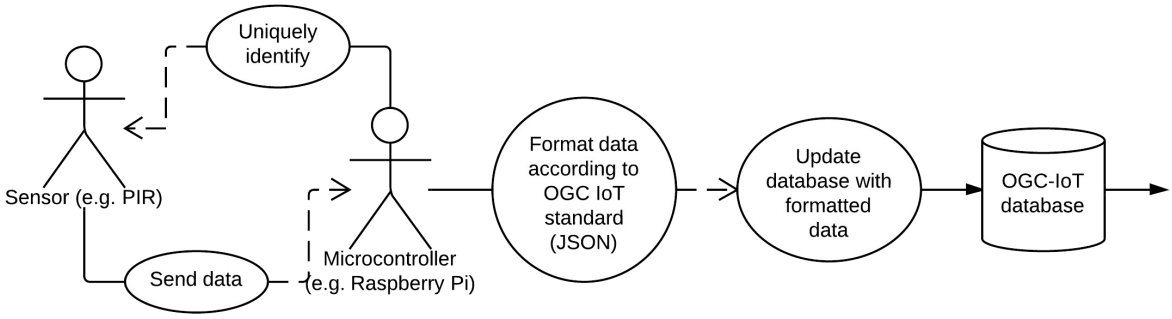

Figure 2: Use Case diagram for Shelf Prototype component of project from Figure 1 above

To briefly explain, we needed to create a specific set of functions and classes that would be able to take interactions from our actors (the Raspberry Pi and the sensors), and subsequently format that data appropriately and upload using the SensorThings API. The specifics as to how we implemented these use cases can be found in other sections, so I will spare you the details here.

The use cases for the website were similar in how we modeled them; however, the actors and their interactions were quite different and in some ways more extensive. Thus we came up with the following use-case diagram in Figure 3:

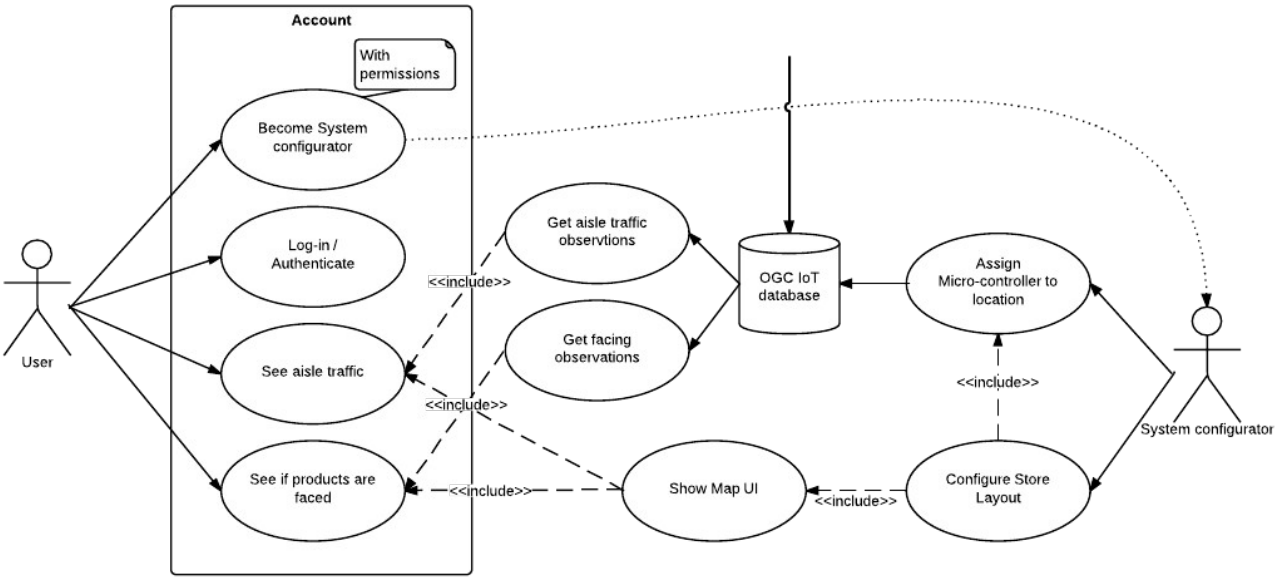

Figure 3: Use Case diagram for Website component of project from Figure 1 above

The primary interactions in the diagram above are our actors (users of the service) interacting with elements on the website. In particular, we specified that our system should have some kind of authentication functionality, as seen inside the boxed area on the left of the diagram. This would allow users to authenticate themselves for their store, and view observations for their store in real time. If users were promoted to “system configurator” status, then they would be allowed to create their own map of their store, as well as assign specific actors from the prototype diagram to locations within their store.

It is important to note the separation of both sides of development, in that the only real dependency between the two sides was the OGC IoT database. This is what allowed our team to split our efforts and have more effective concurrent development. This flexibility helped greatly with developing our final application, as we could decouple development of each of our components from one another. In plain English, this effectively meant that we didn’t need either side to be finished or

working in order to test or debug the other half of the project. Because the SensorThings API allowed us to easily read and update the data on the server, we would be able to push development forward even if we had to simulate data.

Sensors Used

While the technical deliverables report had initially stated that we had determined which sensors we would use and how we would use them, we later found that some of our original ideas would not work so well in practice, particularly due to interference and cost. So while the discussion below doesn't fully match that of our original report, a list of each of the sensors we used for our purposes is described.

Passive or Pseudo-Infrared Sensor (PIR Sensor)

A pseudo-infrared sensor, also referred to as a PIR sensor, is a small, round sensor that measures infrared light radiating from objects within view of the sensor. These types of sensors are often used in motion detectors. A common example of these in use can be seen in the automatic taps and soap dispensers in washrooms. Our aim was to integrate these sensors within a shelf in a store, and track customers throughout the store by mapping the trail of activated motion. The specific brand of PIR sensors we used can be found on sparkfun.com [16]. See Figure 4:

Figure 4: Example Photo of the PIR Sensor [16]

Photo-Interrupter

Initially, we had proposed to check the stock of any particular shelf using magnetometers attached to self-facing shelf units [17]. We envisioned a system that would change the readings on the magnetometer based on the distance that the self-facing unit would have from a "full" position or an "empty" position. This would allow us to know how much stock was available based on the position of the self-facing unit (full or empty or neither).

However, we found a much simpler solution that would not only be cheaper to implement (with respect to raw sensors), but would likewise not require that our shelf-units be self-facing. The solution we discovered was to use photo-interrupter sensors.

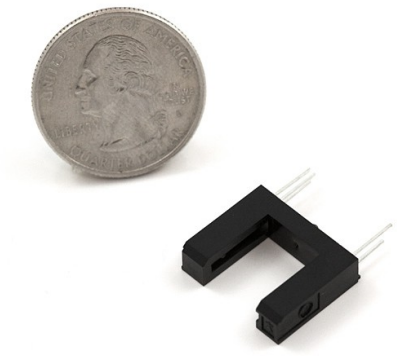

Figure 5: Example of Photo-interrupter from sparkfun.com

The photo-interrupter shown above in Figure 5 above can likewise be found on sparkfun.com [18]. The photo-interrupter works quite simply, in that it produces a true or false value depending on if you block the beam of light between the two bars (shown in the black part of the U shaped device above). In this way, if we separated the two bars by a greater distance, we could place one on each end of a shelf-unit and it would be possible to determine if a shelf had stock or not based on whether the beam was blocked.

Risks Identified

The major risks we identified for the project at this point in time are listed below. Naturally, this was only the first evaluation of our potential risks within the project, so what is listed below is not a full evaluation of all the risks our project entailed. The conclusion describes what we could have, or rather should have done differently in order to more rapidly and accurately identify and mitigate risks within the project.

1. We were concerned that we would not be able to acquire our sensors in time, as we had believed that shipping times might pose a liability. Of course, this would not necessarily impede the progress of the website development, but could pose issues for finishing the prototype development.
2. We felt that there was a chance that we might have ran the risk of developing the two sides of the project orthogonally to one another if we did not have strict adherence to the use cases. Fortunately our use-cases were well thought out at the time of writing, so we eventually found this to not be as big of an issue as we might have originally anticipated.
3. While we did do some cost analysis regarding the sensors, we didn't do an extensive study into what the typical cost might be for the end-product we wished to develop. Naturally, the cost of the sensors, while cheap, grows as more sensors are needed. However, we never identified the full cost per unit at that point in time, and ran the risk of having a project that others wouldn't try to use or develop off of if the cost of development caused a significant barrier to entry.
4. The group had concerns that when the sensors did ship, they might not function as expected, or they might be too sensitive to changes in the environment. For this reason, we felt that a certain amount of time would be required in order to properly calibrate these sensors.
5. Implementing the user system as described in the use case proved to be challenging, so we did potentially risk not having a fully fleshed out user system as described

above. In the end, we found that adding user-tiers (regular user vs. system configurator) was too challenging and did not really add a significant return for the investment. Put plainly, even if we had different levels for users, the overall functionality of the project as a whole would not change. Additionally, as one of the objectives of our project was to help provide an example to help proliferate the OGC SensorThings API, the loss incurred by not implementing such a system would not significantly affect the outcome of this goal.

6. One of the most significant problems that we could rarely test was transferring large amounts of data to and from the SensorThings API server. When we initially started using the test server provided for us, we found that it reset itself quite frequently, which prevented us from being able to scale the project up to many sensors and observations. In the end, some specific limitations of the server were discovered, namely that it doesn't properly return `last-modified` fields in the HTTP headers, which created a huge inefficiency every time we tried to download and test data (since we couldn't properly obtain HTTP 304 response codes). There were some other minor issues, but those will be discussed later. After identifying these issues and bringing them up with our supervisor, we managed to mitigate the largest challenges and still ship a working prototype of the entire system.

As always, the full reports are available in Appendix A [14] [19].

Prototype Design and Usage

Hardware Components

Breadboard

A breadboard is used to build and test circuits quickly before finalizing any circuit design. The Breadboard has many holes into which components like wires, resistors, pi cobblers can be inserted. The breadboard that was used for this project is shown in Figure 6 below.

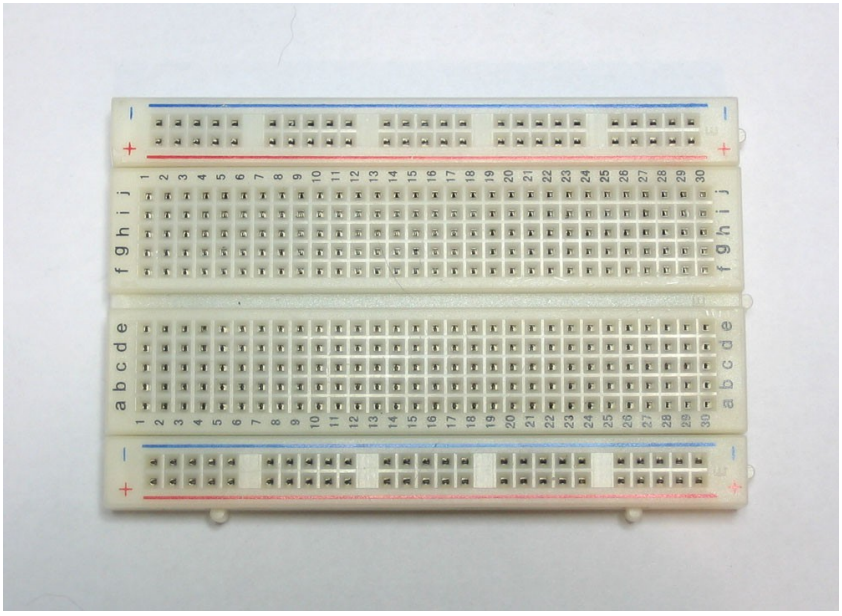

Figure 6: Example of the breadboard we used for testing

Since there are 5 clips on one strip, you can connect up to 5 components. As you can see there are 10 clips on the strip and it is separated by a ravine. The ravine separates the 2 sides and they are not

electrically connected. Tutorials made available through sparkfun.com were used to better understand how the breadboard works and it has proven to be very helpful.

Pi Cobbler

Though the Raspberry Pi comes with GPIO pins directly available on the board, a Pi Cobbler is very convenient and useful if you have a lot of connections to make. The Pi Cobbler that we used for this project is shown in Figure 7:

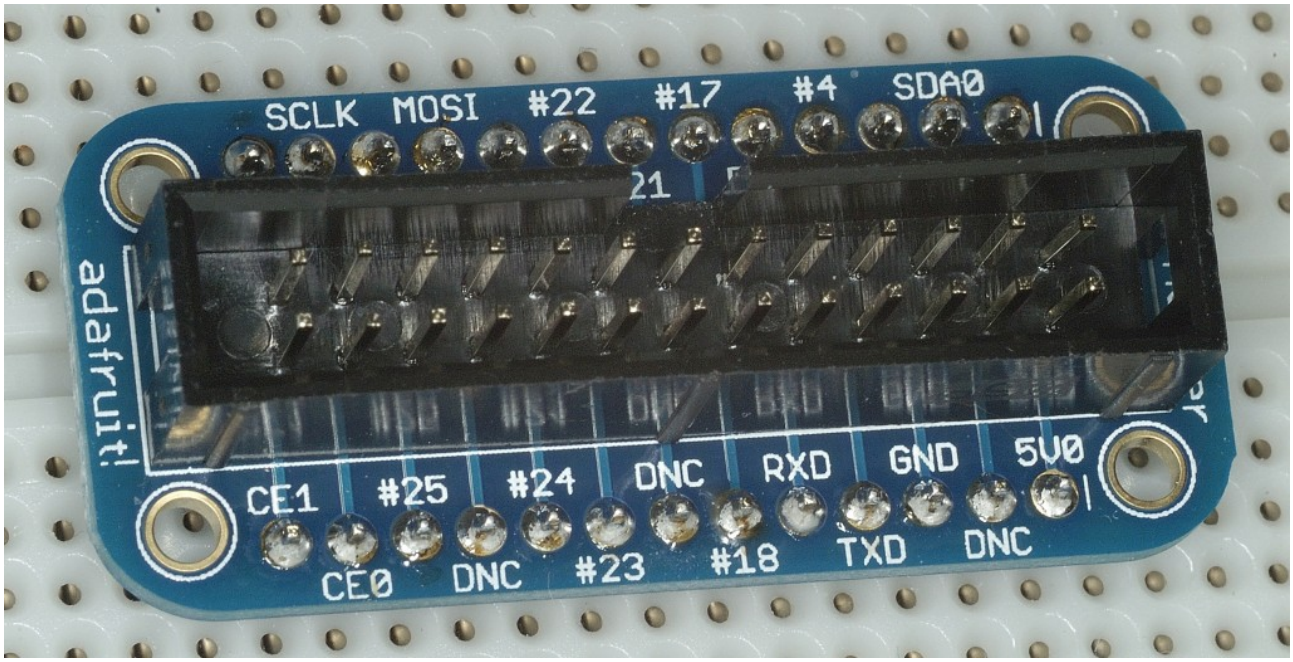

Figure 7: An example of the Pi Cobbler, an accessory for the Raspberry Pi

It is also necessary to follow the GPIO layout map, shown below, to make sure that you are using the correct pins and labels, even when using the Pi Cobbler. There are 5 pins that can be connected to the ground, 2 that can be connected to 5V and 14 pins that can be used as GPIO. A python code is used to reference pins by their labels. If something is labelled incorrectly then the code will produce many errors. We had a hard time understanding how the GPIO pins worked, but the diagram (shown in Figure 8) has helped us a lot.

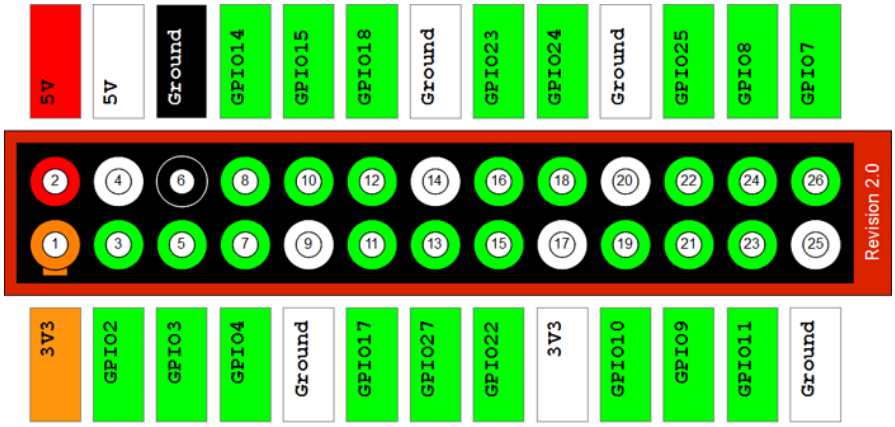

Figure 8: Layout map of the gpio pins on the Raspberry Pi and Pi Cobbler

PIR Motion Sensor

One of the main focuses on the hardware side of LASS involved setting up a PIR sensor to track

customer movement. In this section we will look at what we did to include this sensor in our prototype, in terms of both wiring and Python code.

Wiring the Sensor

Since none of us had much experience with electronics when we started this project, the first thing we learned was to stick to your datasheet [20]. The datasheet provides specific information for your sensor and how to use it without burning the place down.

For the PIR, you will notice that it has three connections: DC 12V, Alarm, and Ground. Using a Raspberry Pi, a breakout cable and a breadboard, we were able to simply connect each of these to their respective pins as shown below. For the alarm connection, we used GPIO Pin #18. See Figure 9 below:

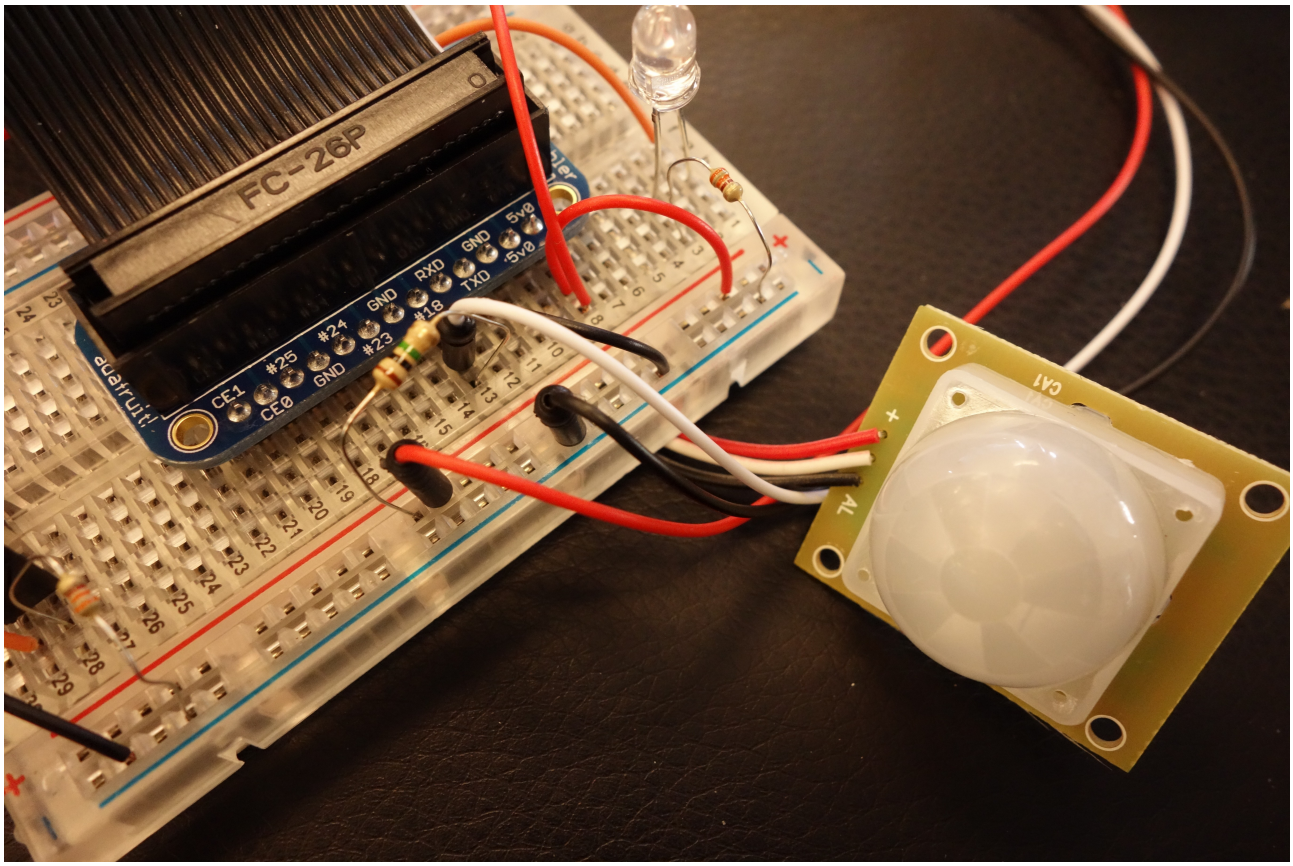

Figure 9: Example of how wiring is set up for PIR Motion Sensor

Coding in Python

Once the sensor was set up, something had to read and record the data once it reached the RaspberryPi on the other side of the breakout cable.

For this we implemented a short Python script, excerpted below, beginning with the inclusion of the GPIO library, the pin set-up, and a quick connection check. It is also necessary to specify the numbering mode of the GPIO pin, since there exists more than one. We used BCM, or Broadcom.

```
import RPi.GPIO as GPI
GPIO.setmode(GPIO.BCM)
GPIO_PIR = 18

GPIO.setup(GPIO_PIR,GPIO.IN)
```

```
if GPIO.input(18):
    print('Port 18 is 1/GPIO.HIGH/True')
else:
    print('Port 18 is 0/GPIO.LOW/False')
```

If the input of GPIO #18 is 1, then the sensor has been activated. Otherwise, it is zero. To continuously check the status of the sensor, we used to following logic contained within a while loop:

```
try:
    print "Waiting for PIR to settle ..."
    while GPIO.input(GPIO_PIR)==0:
        Current_State  = 1
    while True :
            # Read PIR state
            Current_State = GPIO.input(GPIO_PIR)

            if Current_State==0 and Previous_State==1:
                # PIR is triggered
                Previous_State=0
            elif Current_State==1 and Previous_State==0:
                # PIR has returned to ready state
                Previous_State=1
        time.sleep(0.01)
except KeyboardInterrupt:
GPIO.cleanup()
```

Since we are sending observations to the server in real time, this code allows us to send an observation only when the PIR transitions between steady and triggered states. Since the sleep time is so small, it helps us avoid constant triggers during high movement periods or periods of complete rest, while still posting in real time.

Issues Encountered

The PIR sensor turned out to be extremely sensitive, to the point where it would be constantly triggered by nothing at all. We managed to calm it down by placing a resistor between the voltage and alarm connections. You can see it in the above picture.

Photo-interrupter

In addition to tracking customer movement, we also wanted to be able to tell if the shelf was ‘faced’ or not. We chose to use a Photo Interrupter to complete this task.

This small sensor has five pins and sits directly in the breadboard. It works by shooting a beam of light between its two prongs. When an object is placed between the prongs, the path of light is blocked, and the detector reads zero. Otherwise, it reads one.

Wiring the Sensor

Since the Photo Interrupter is pretty much symmetrical, it is important to note that the data sheet diagram shows the sensor FROM ABOVE (as confirmed by the small and barely noticeable capacitor icon on the tip of one if the prongs). Mixing this up has the potential to result in roasted equipment. See Figure 10

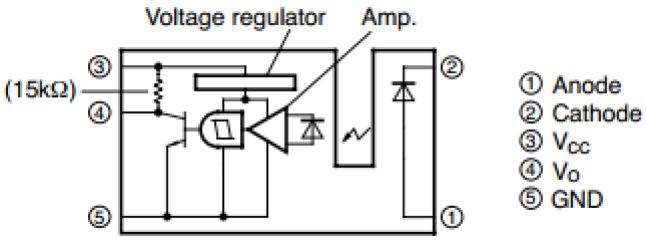

Figure 10: Schematic Diagram of Photo-interrupter

Another interesting thing is that both sides of the Photo Interrupter are wired individually. This gives us the option of splitting it in half to widen the gap between prongs if need be.

Since each side is independent (one is a simple capacitor), the sensor has two ground connections (one with a resistor) and two voltage connections. On the detector side, there is an alarm connection, which we attached to GPIO # 17.

Coding in Python

As in the previous section, the GPIO pins were set up and tested:

```
import RPi.GPIO as GPI
GPIO.setmode(GPIO.BCM)
GPIO_Pint = 17

GPIO.setup(GPIO_Pint,GPIO.IN)

if GPIO.input(17):
    print('Port 17 is 1/GPIO.HIGH/True')
else:
    print('Port 17 is 0/GPIO.LOW/False')
```

This time, the steady state of the sensor would not rest at zero, but at one. Using a while loop to continuously check the sensor status, we used the following logic to isolate instances in which the sensor transitions from steady state to triggered state. Only under these conditions will it attempt to post an observation to the server (see following section below for more information). This eliminates the constant stream of zeros that it would otherwise post once it is triggered.

```
Switch_State = 1
Prev_Switch_State = 1
try:
    Switch_State = GPIO.input(17)
        if Switch_State == 0 and PSS == 1:
            #SWITCH STATE IS ZERO
        #post obs
            Prev_Switch_State = 0
        elif Switch_State == 1 and PSS == 0:
            #Switch is reset
        #post obs
            PSS = 1
     time.sleep(0.01)
except KeyboardInterrupt:
GPIO.cleanup()
```

And not only that – it allows us to post both possible instances in time: when the switch was triggered, and when it returned to steady state. In other words, when the shelf became un-faced and when it was stocked up again. Both of these transitions of state would be valuable information to a store owner.

LEDs for POSTing Observations

One condition we wished to satisfy for debugging purposes was setting an LED to blink once an observation was confirmed to be posted. We managed this by connecting an LED to a GPIO pin on the Raspberry Pi, with wiring similar to the diagram below. The resistor used was 330 ohms. It is important to use the 330 Ohm resistor only because otherwise it would give you many errors. See the schematic diagram in Figure 11:

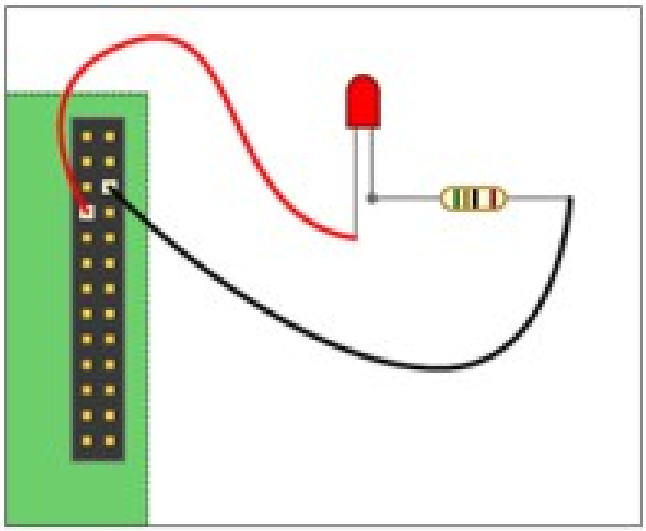

Figure 11: Schematic of how we wired the LED with a resistor

The GPIO pin used was pin #4 in terms BCM references, and it was set up in much the same way as the GPIO pins #17 and #18, except it was set to output rather than input.

After that, all we had to do was tell it when to light up using the following line:

```
GPIO.output(GPIO_LED,True)
```

The blinking of the LED is positioned effectively in the script to ensure that the prototype is not only collecting data, but also posting it to the SensorThings API data service as well. See Figure 12.

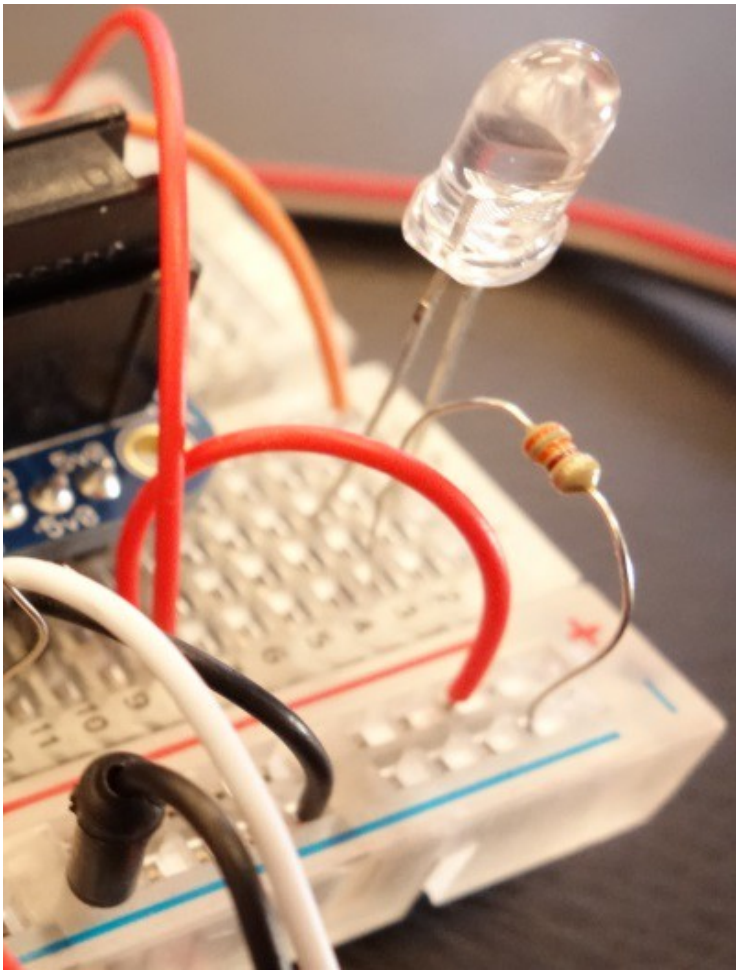

Figure 12: Photo of how we set up the LED on the breadboard

Putting It All Together

Overall, several of these components were connected together, using the breadboard and Raspberry Pi as the central pieces. Currently, we have three Raspberry Pi's setup for collecting and sending data. Each Raspberry Pi has two sensors connected to it. The equipment that is currently being used are 3 Raspberry Pi's, 3 Photo Interrupters, 3 PIR Motion Sensors, 3 LED's, 3 Breakout cables and a few wires. The functionality of the LED on the breadboard is that if there is motion detected or if there is an object in between the sensor, the LED light goes off; however, this was primarily for debugging purposes and will likely not be seen in future work on the project. See Figure 13 below for an example of the entire prototype put together:

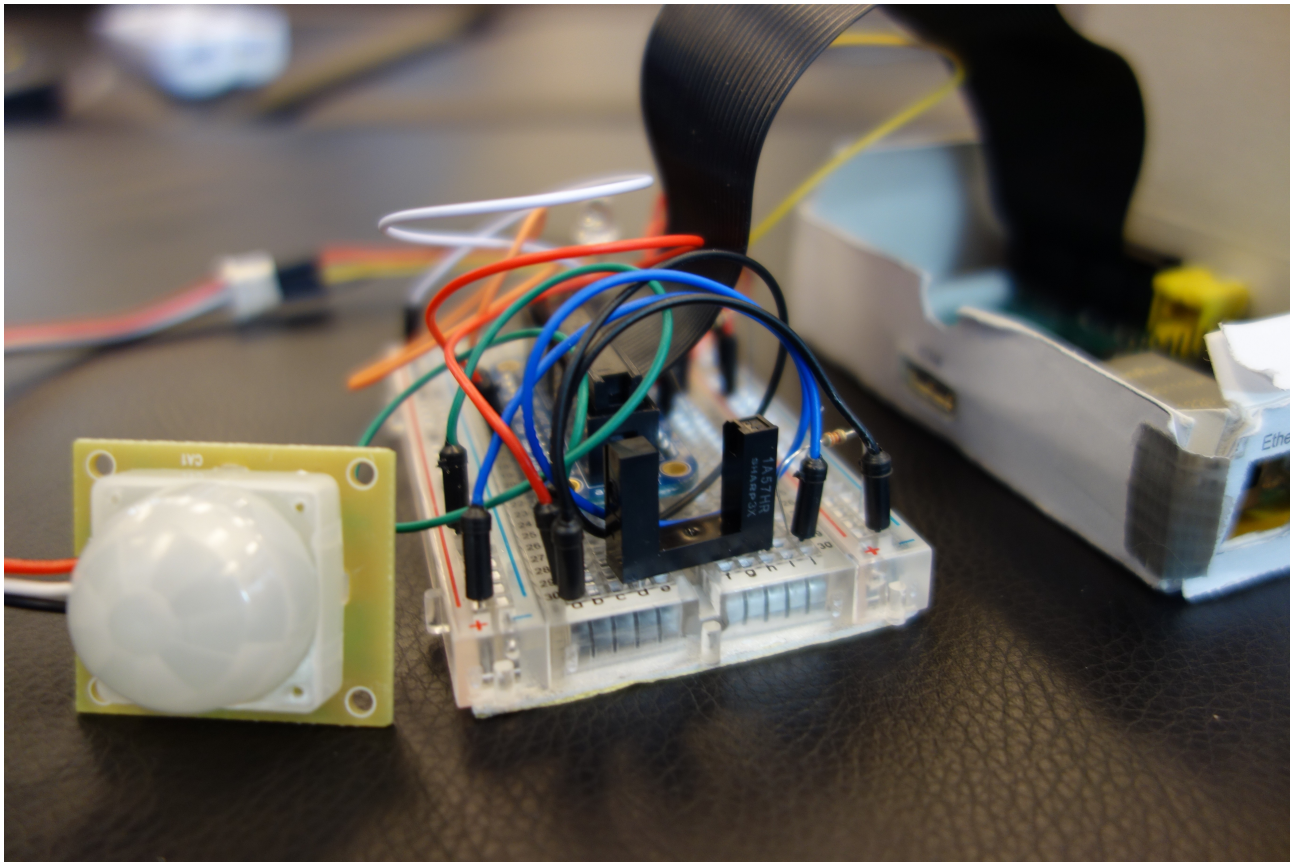

Figure 13: Example set up of all the sensors and components together

Software Components

Understanding and Using the OGC SensorThings Data Model

The OGC SensorThings API has a general data model, designed to be adaptable to any IoT application. This post will discuss the components of the data model which we deemed relevant for our project, and how these components were used in our application. For more details regarding other applications, please see the full documentation on the OGC IoT data model page [1].

As you can see in the figure below (Figure 14), we have highlighted the entities in the data model which were the most crucial for our application.

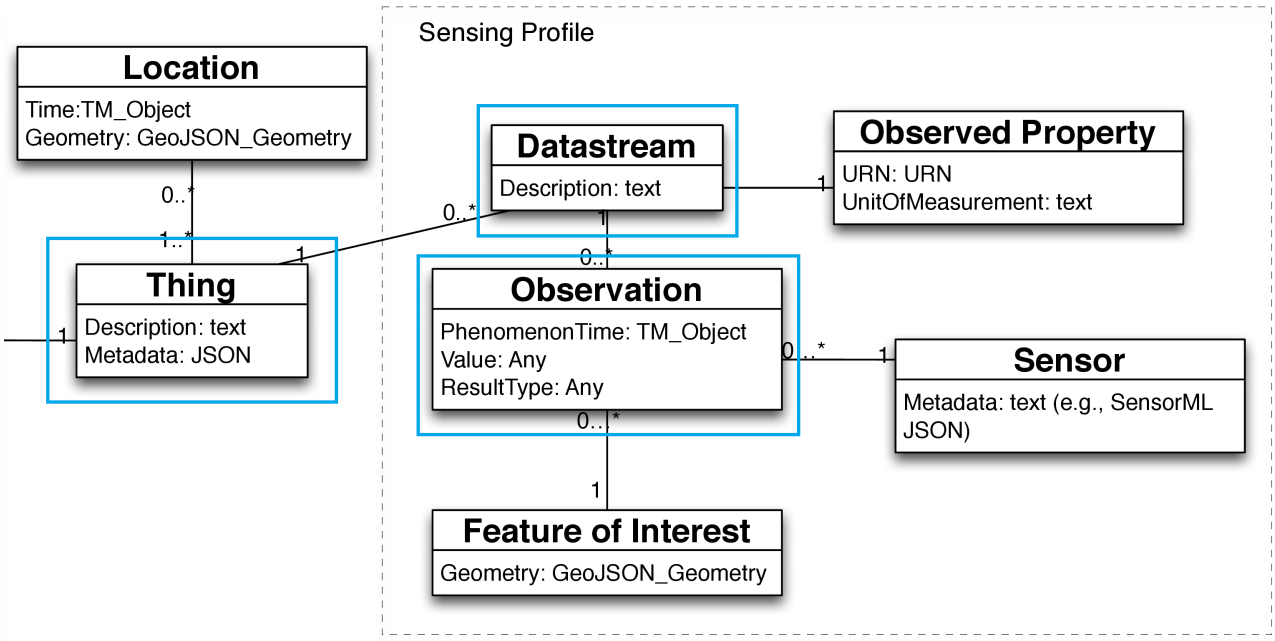

Figure 14: Data model defined by the OGC

The core of this data model is the “Thing”, which usually corresponds to some real-world object. In the case of LASS, the Thing should be a shelf; however, in the proof-of-concept phase, we decided to identify the Raspberry Pi as our Thing.

From this Thing, there can be 0 to many associated Datastreams. A Datastream is an entity which groups observations together. Since we were interested in tracking two main types of observations, we likewise had two Datastreams.

Finally, each of these associated Datastreams have 0 to many Observations, which are direct readings from the sensors. Both of our sensors operate as “on/off”-type switches, so our Observation values were either 0 or 1.

All this information is set up and sent from the Raspberry Pi to the OGC IoT data service, which in turn was read from our website. The diagram below shows a high-level overview of our project, with the appropriate entities from the SensorThings API Data Model applied. See Figure 15 below as a visualization of how we applied the data model to our design.

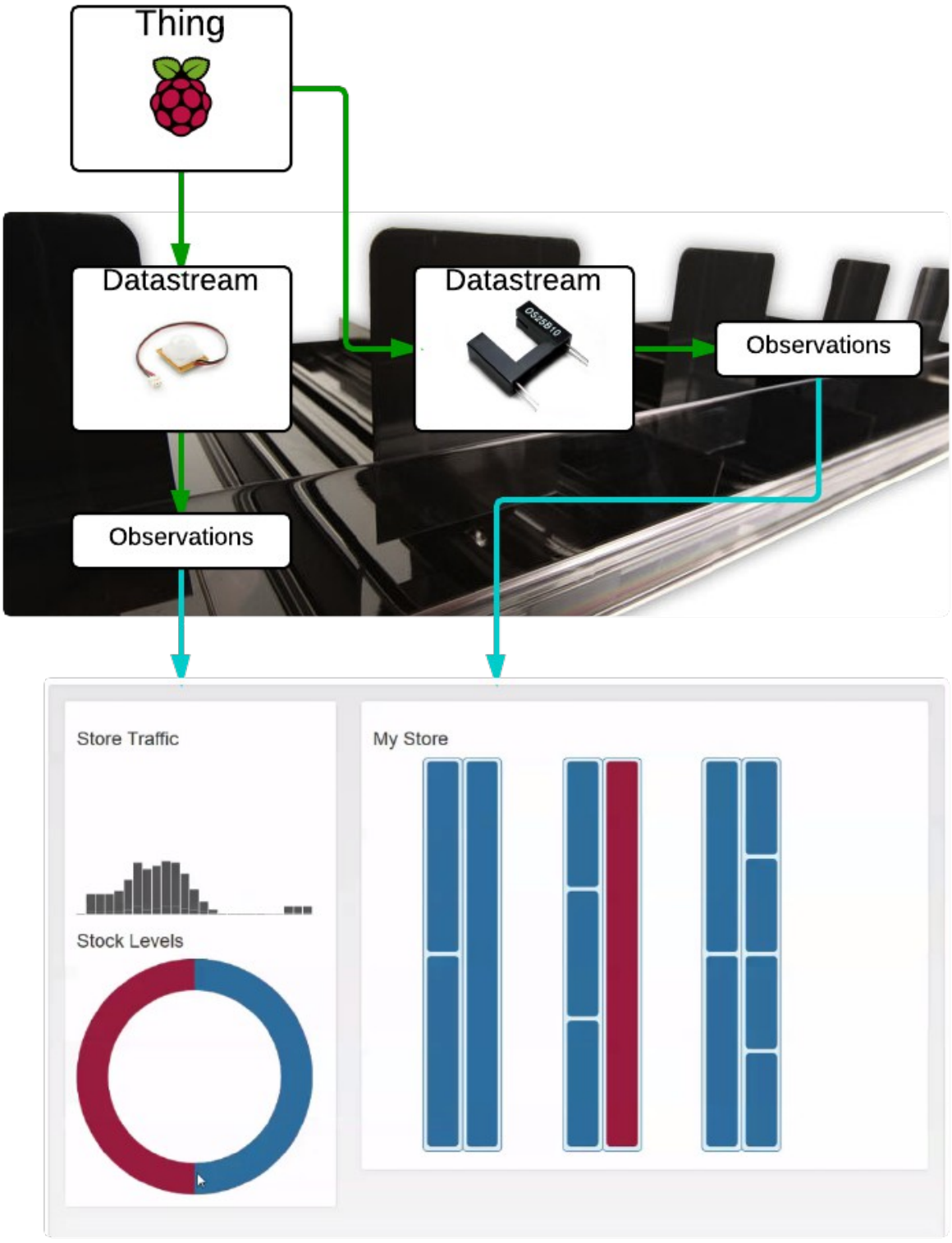

Figure 15: Adopted data model we used for our design

Using the OGC IoT SensorThings API

This section attempts to explain how to use HTTP POST (corresponding to the “CREATE” in Create, Read, Update, Delete) to send data to the OGC IoT data service. You can find further

documentation here at their website [1]. There is a handy quickstart manual provided alongside their Interactive SDK, which shows examples in terms of how to format your request.

In order to describe how to use the API, we will explore an example of making a simple POST request and create a new Thing. For the purposes of experimenting and debugging, using Hurl [21] is recommended.

We'll start by changing GET in the drop down menu to POST. Following that, we can put the following URL in the text box:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things
```

This is an example URL to access the "Things" collection in the data service. Next, we need to add a new header (which can be done by clicking on the +Add Header(s) button if using Hurl). The "name" field should be Content-Type and the "value" field should be application/json.

Finally, we can add the content we would like to post. We will keep it simple for now, and create our Thing with just a description. Something like this will suffice:

```
{"Description":"This is a chair"}
```

Your final screen will look something like in Figure 16:

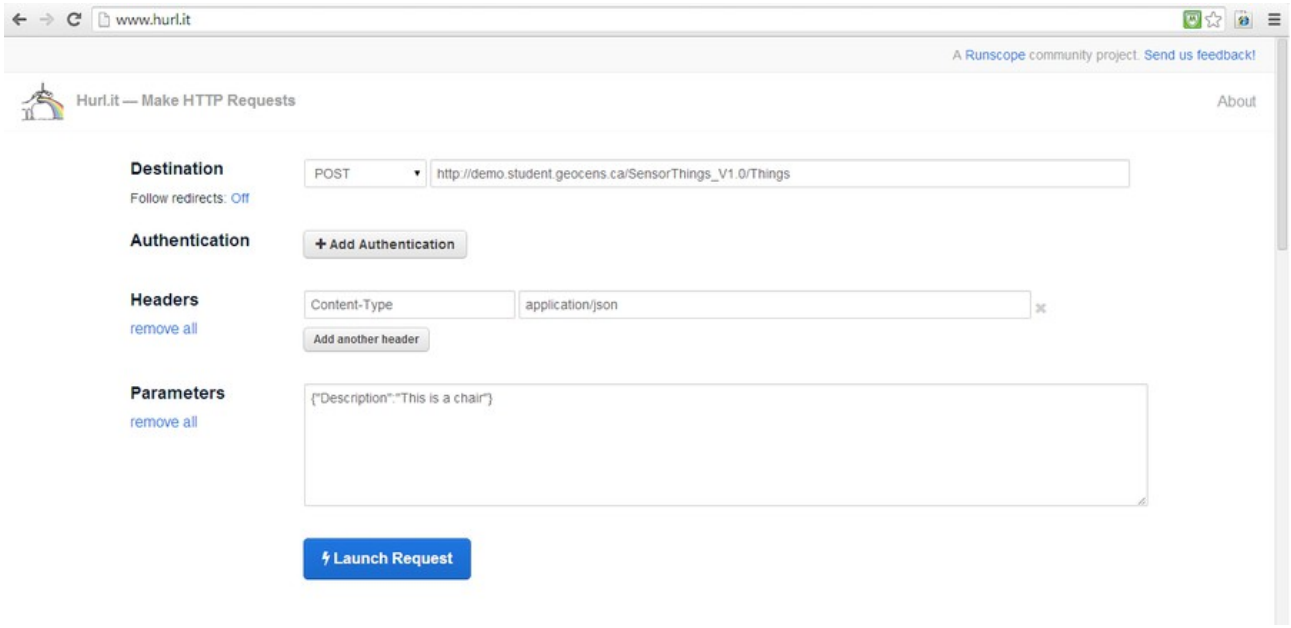

Figure 16: Screenshot of what the hurl request should appear like

Once you send it (just click Launch Request), the response you will get is 201 Created (see Figure 17 below):

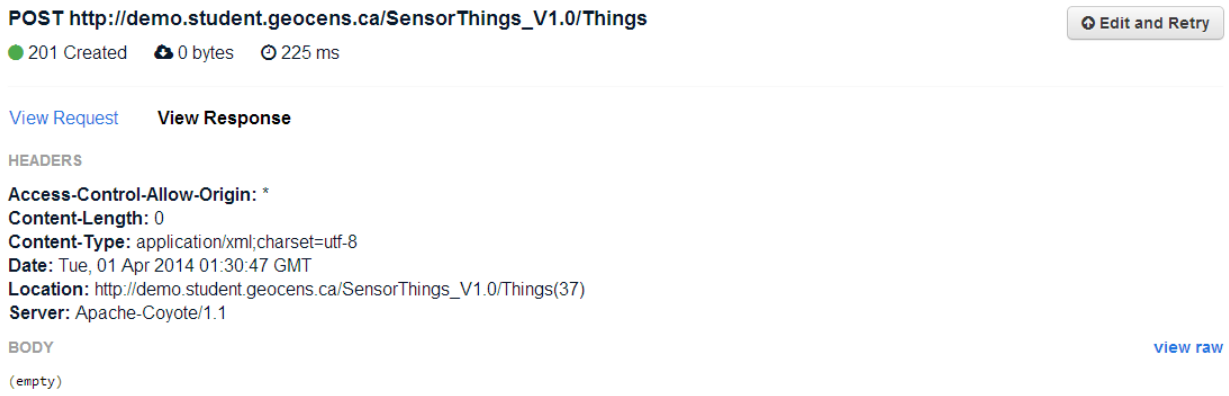

Figure 17: Result from sending a POST request to the SensorThings API

POST-ing information from the Raspberry Pi

In the section above, we discussed in general how to interact with the SensorThings API. The intent is to send data to the server as per our use case outlined in the *Design Process – Technical Deliverables & Progress Report* section above. Now we'll look at the details for how we formatted and sent our actual data from our shelf prototype system.

A summary of the procedure is as follows:

1. Create Thing with associated Datastreams
2. Obtain URI (Unique Resource Identifier) of Thing
3. Obtain assigned Datastream IDs
4. Post Observations, linked to the appropriate Datastream

Step 1: Create 'Thing'

Before sending observations to the data service, it's necessary to create the appropriate Thing and Datastream. After this has been created, it doesn't need to be created again.

To create the Thing, you need to make a POST request to the Things collection at the root URI, which is the online address for the SensorThings data service:

```
url = http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things
```

The body of the POST request should appear as follows:

```
postBody = {
    'Description': 'This is a Raspberry Pi on a shelf',
    'Datastreams': [
         {'Description': 'This is a datastream for measuring people traffic'},
         {'Description': 'This is a datastream for measuring shelf stock'}
    ]
}
```

If you're using Python 2.7 (as we were), you can use the Requests library [22], which makes HTTP requests quite intuitive. It will look something like this:

```
headers = {'content-type': 'application/json'}
r = requests.post(url, data = json.dumps(postBody), headers = headers)
```

(You'll need to import both the requests and built-in json libraries)

After sending this request, we should have successfully created a new Thing with two Datastreams. Next we need to find where it's located, so that we can eventually send Observations.

Step 2: Obtain URI of Thing

You'll notice that we created a variable "r" along with our post request. Once the request is processed, "r" contains our response. We can find where our Thing is using the following code:

```
thing_location = r.headers('location')
```

Maybe you're wondering – "Why do we have to do this at all? Can't we assign it a specific location?"

Well, the easy answer is this: the way that the data service is currently set up, you don't assign your Thing an ID. You create a Thing, and its ID will automatically be assigned (for example, if there are already 7 Things on the server, your Thing will have ID = 8). So no, you cannot assign a specific location or ID.

Your thing_location, assuming an ID of 8, should have a format something like this:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things(8)
```

Step 3: Obtain assigned Datastream IDs

Just as Things are automatically assigned IDs, Datastreams are also assigned IDs. Because each Thing has its own associated Datastreams, we can take a look at all of them just by visiting this URL:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things(8)/Datastreams
```

Programmatically speaking, we can get our Datastream IDs if we know how many Datastreams we have (which we should – we made them when we made our Thing). In our implementation, we achieved this with the getDatastreamID function:

```
def getDatastreamID(self, dsIndex):
    url= self.thing_location + '/Datastreams'
    r = requests.get(url)
    response = r.json()
    datastreamID = response['Datastreams'][dsIndex]['ID']
    return datastreamID
```

dsIndex would be, in our case, either 0 or 1: 0 corresponds to the motion sensor and 1 corresponds to the photo interrupter (stock sensor).

Step 4: POST Observations

This final step is the only one that should be executed more than once. Now that everything has been set up, we can post observations as we read them in real time. They will be posted to the Observations collection, for example:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Observations
```

In order to attach the Observations to the proper Datastream, this should be included in the body of response. An example Observation post in the program is shown below:

```
payloadOBS = {
```

```
        'Time':         time.strftime('%FT%T%z'),
        'ResultValue':  '1',
        'ResultType':   'Measure',
        'Datastream':   {'ID':datastreamID},
        'Sensor':       {'ID':self.sensorID}
    }
```

For our observations, the Time property is created such that it matches the ISO 8601 date format. The ResultValue is either 0 or 1, the ResultType is always ‘Measure’, and the ‘Datastream’ ID is obtained from step 3.

What’s this ‘Sensor’ property about? Based on the current data model, an Observation should link to an associated Sensor entity. You can create a Sensor ID in a similar way as you created a Thing, except you would POST the request to the /Sensors collection. You can try it yourself, and if you’re not sure, check out our source code in our GitHub repository [23].

Allowing the Prototype to Run Without User Interaction

As much as we enjoyed playing with sensors, the LASS prototype wasn’t supposed to be used only by the developers. This section considers the steps we took to make our prototype as hands free as possible, so that the user – potentially a store manager – can just plug it in and go.

During the development process, we found the most efficient way to get things done was to write code directly onto the RaspberryPi itself. This allowed us to test the sensor functions and GPIO libraries as we went along, but also required us to view the display via HDMI cable.

Now that the sensors had been tested and are fully operational, there is no longer a need to view the screen or interact with the interface at all, on the conditions that we can run the code automatically and view the status of the readings (see: why we used the LED for development above).

Running at boot time

To run the sensor scripts automatically without help of any screen, mouse, or keyboard, we set the scripts to run on start up. This was completed via the following steps in terminal:

Create a folder in which to store the auto running script

```
$ mkdir ~/bin
$ cd ~/bin
```

Create and edit the script as root:

```
# nano script_auto_run
```

The script should read:

```
#!/bin/bash
# Script to start our application
echo "Doing autorun script..."
sudo /path/to/script &  (If you wanted to add a python file, the last line would
read "sudo python /path/...")
```

Make the script executable (again, as root)

```
# chmod +x script_auto_run
```

Lastly, make the following edits as root to /etc/rc.local

```
# nano /etc/rc.local
```

add the following line to the end of the file:

```
/home/pi/bin/script_auto_run
```

Saving the path of script_auto_run in this file will run the application on start up.

Use Cases Implemented

As mentioned in the design process above, there were specific use cases (outlined in Figure 2) that needed to be completed to make our "shelf", as it were, functional. These use cases stemmed directly from our functional requirements for the project, and the completion of such use cases is outlined below.

1.1 – Update the Database With Formatted Data

Once the hardware is set up and working (see use cases 1.4 and 1.2), the database should be accessed and updated with the new, formatted data. This was to be implemented in real time.

This use case was completed by sending observations with the `sendObs()` class entity that we defined as shown in use case 1.2. It takes in two arguments: the observation, which in our prototype can only be a one or a zero, and the datastream identity (which corresponds to the sensor from which we would like to post). The `sendObs()` entity also formats the observation as specified by the data model and by use case 1.2.

1.2 – Format Data According to the OGC IoT Standard

Data collected by the sensors and passed to the microcontroller (use case 1.4) was to be formatted to enable use case 1.1.

Below is the `sendObs()` class entity as discussed in use case 1.1. In addition to posting the observation integer to a datastream, it also formats the observation as required by the data model as well as this use case.

```
def sendObs(self, obs, datastreamID):
    urlOBS = self.rootURI() + 'Observations'
    headers = {'content-type': 'application/json'}
    payloadOBS = {
        'Time':time.strftime('%FT%T%z'),
        'ResultValue':str(obs),
        'ResultType':'Measure',
        'Datastream':{'ID':datastreamID},
        'Sensor' : {'ID' : self.sensorID}
    }
```

1.3 – Identify Sensors

The sensors and microcontrollers were to be recognized and given a unique ID, so that their datastreams may be distinguished even after the data is formatted and updated to the database.

The microcontrollers were given their own names during setup, but were also assigned a specific object number as set out by the OGC SensorThingsAPI general data model. Each microcontroller became a separate `thing` entity in the model.

The general data model also permitted the creation of several `datastreams` for each `thing`, so that multiple sensor data from the same microcontroller is distinguishable. The following demonstrates the creation of a `thing` with two datastreams.

```
url = self.rootURI() + 'Things'
payload = {
    'Description': 'This is the real TurboCat (Raspberry Pi)',
    'Datastreams': [
        {'Description': 'This is a datastream for measuring people traffic'},
        {'Description': 'This is for measuring shelf facing'}
    ]
}
headers = {'content-type': 'application/json'}
```

To avoid creating a new set of `things` and `datastreams` every time we ran the prototype, we implemented a little work-around that would store the previous 'thing' location (just a URL) on the Raspberry Pi in a text file. When running the prototype, it would first check these files for the URLs of possible existing `things`.

```
with open(URLtextfile.txt, 'a+') as x:
    URL = x.read(starting point)
    if (URL == text):
        # thing exists
        # check response from URL in file
        # if response = 200, use URL, continue
        # if response = 404, wipe txt file, create new thing
    else:
        # thing does not exist.
        # create thing to identify microcontroller
#continue with script
```

As shown by the pseudo code above, if it finds that a URL exists and does not return a 404 error, it will continue to identify the microcontroller as that `thing`. Otherwise, if the URL does not exist, it will create a new `thing` to identify the microcontroller. In the case that the response of an existing URL returns a 404 error, the text files are wiped clean and a new `thing` is created. This feature was particularly useful during the testing phase in which the webserver we were provided with was being intermittently reset.

1.4 – Send Data to the Micro-Controller

Observations from a sensor will be acknowledged by the microcontroller and stored for use cases 1.2 and 1.1.

This use case was fulfilled through the functionality provide by the GPIO library and sensor functions. Details can be viewed in previous sections about the sensors.

Website / Interface Development

Webserver Design

Basics

A critical component of our website development revolved around building a webserver that could render pages and serve content dynamically. What's more, we wanted to develop our server such that everybody involved in the project could set it up easily with minimal barriers for development. Currently, the code for the webserver has been split off from the main repository and is now hosted in the ENGO500-Webserver repostitory [5]. This section will take you through how to install the pre-requisites and get the server up and running on your local machine.

The first design decision we made when choosing how to develop our server was to choose a language or framework to write it in. Ultimately, we decided to develop the server application using

Node.js [24]. We chose node for a number of reasons, but it basically broke down to the following:

1. Node.js is basically just JavaScript. Since a large portion of the frontend development involved jQuery, D3.js, and other JavaScript functionality, we thought it would be best to use Node.js since it meant we could develop in a single language for both the server and the client.
2. Using the Express.js framework [25] with Node made it really easy to set up our server. Beyond that, it is even easy enough to understand for people who are beginners at JavaScript, which was an unintentional side-effect we were very happy to run into. Effectively, we found that this lowered the barrier to entry to our server development, which makes it easy to hack and tinker with.
3. The Node.js community is thriving, and the number of packages available in the Node Package Manager (NPM) is already quite extensive and growing. This made it easier to implement some of the more difficult components of the server (such as user authentication), by providing extensible modules with which we could develop on top of.

Secondly, we needed something with which to handle our user authentication / storage system. In this sense, we don't need to store data about the sensors, since that is handled by the SensorThings API service. Instead, we need to store user preferences and password data. For this, we decided on using MongoDB [26], handled by the Mongoose [27]module provided by NPM.

Together, these are the major software dependencies of our project (barring git [http://git-scm.com], which we highly recommend using, since our project is tracked using git and hosted on Github). Most of these are incredibly easy to set up and install, which will be explained in the following section.

Installing Pre-requisites

Installing Node.js can be managed in a few ways. The installation process itself varies based on the operating system that you use, but the documentation at nodejs.org is quite good. While the project itself was mostly developed using Node v0.10.25 and v0.10.26, it should for the most part be forwards compatible with newer versions of node. If there are any troubles, please feel free to contact anyone on our team and we'll try our best to help you. If you're on a Mac OS X machine or a Linux machine, I highly recommend using the Node Version Manager (NVM), which makes managing your Node.js installation much, much easier.

Installing MongoDB can be a bit more of a beast than installing Node.js. In particular, MongoDB presents particular troubles when installing on Windows, but has similar issues when installing from outside the package manager on Mac or Linux. For this reason, we provided a script in the ENGO500-Webserver repository in the scripts/ folder, to make this easier. The particular issues with how Mongo distributes binaries in a zip archive aside, if you're able to use any version higher than 2.4, you should be set to use what we've built for this project. If you are on Mac OS X or any Linux variant, I highly suggest the use of a package manager to install MongoDB, unless you know what you're doing. If you don't have a package manager on Mac OS X, I highly recommend Homebrew (http://brew.sh/). A simple `brew install mongodb` should suffice, but please check the documentation first.

Running the Server

Great, so now I'll assume that you have both of the major pre-requisites installed, it should be easy enough to get the project up and running. The following few commands are copied from the ENGO500-Webserver `README.md` file, but basically you'll just want to:

```
$ git clone https://github.com/ThatGeoGuy/ENGO500-Webserver.git
$ cd ENGO500-Webserver
$ npm install            # Installs extra node.js modules
```

And just like that, the server should now have all the current modules installed. Running the server has one more step involved, but it is a simple one. Since MongoDB needs to be told *where* to start saving the database, we'll have to specify it to start before we run the `server.js` app in Node. For the purposes of development up until now, we've used a folder called `mongo/` within our repository (don't worry, git will ignore your database) on port 29999 to store the database. In a separate terminal window, in the ENGO500-Webserver repository, run:

```
$ mkdir mongo/
$ mongod --dbpath mongo/ --port 29999
```

Alternatively, if you don't mind starting with a fresh database everytime you want to run the server, you can run `./scripts/startServer.sh` (or `./scripts/startServer.bat` if you're on Windows) which will remove the `mongo/` directory before running the two above commands again.

Afterwards, run the following command in a separate terminal window:

```
$ node server.js
```

And voila! The website should be available on http://localhost:8000/.

Server Configuration

This section discusses how the code and structure of the server is laid out in the ENGO500-Webserver repository. Hopefully, this article will provide some insight into how the project was structured from a software point of view, and likewise should give you enough information to hack and modify the webserver to suit your own needs.

Directory Structure

The directory structure of the repository is meant to be straightforward, and follows a typical Model-View-Controller (MVC) style. A tree structure of the directories in the repository is shown, along with a short descripiton of what each directory represents and what kinds of files are inside of them.

```
ENGO500-Webserver/
├── config
├── models
├── public
│   ├── css
│   ├── img
│   │   └── icons
│   └── js
├── routes
├── scripts
└── views
```

Root Folder (ENGO500-Webserver/)

The root folder (presumably named ENGO500-Webserver) contains the whole of the project. Mostly, files are split up amongst the other folders within the repository, but one file in particular is important in this directory: `server.js`

The `server.js` file is important because it defines the very base commands for configuring the server. If you were to download and start using this project with the intent of modifying it, starting

with this file would be your best bet. The reason for that is that `server.js` is the most central file to the entire system. With regards to our MVC model, the `server.js` is the beginning of our *controllers*.

Routes and Route Handlers (routes/)

The purpose of the routes folder is primarily an extension of the `server.js` file found in the root folder. Within this directory there are two files of particular importance: `getHandlers.js` and `postHandlers.js`, which comprise functions and route definitions for the website for both HTTP GET requests (`getHandlers.js`) and HTTP POST requests (`postHandlers.js`). Again, these files constitute functionality that corresponds to the *controller* part of our MVC framework. If you're planning on adding a page to the server, or planning on studying / changing how information is passed to the server, the two files in this folder will be where you want to make your changes.

If you look at `getHandlers.js` specifically, you'll notice that it (much like it's counterpart `postHandlers.js`) is a module that returns a function. The two arguments for the function are the Express.js application variable (aptly named `app`) and a variable which holds the functionality to user-authentication for the server. This functionality will be explained in a later post, but for now let's get back to how `getHandlers.js` works. For each HTTP GET request that gets sent to the server, a route and handler will be specified in the following way:

```
app.get(route, handler);
```

Where route is the name of the path on the site (e.g. /login), and handler is a callback function that takes in two arguments (a request and a response) and tells the application what to do once a request is found for that route. Since `server.js` has been configured to allow for Nunjucks templating [28], it's easy to return a simple template rendered from our views in our MVC framework.

Views (views/)

As could be expected, the `views/` folder represents the *view* portion of our MVC framework. As mentioned previously, we used Nunjucks (Jinja2) style templates, from within the Express.js framework. Effectively this means that our views consist of a series of templates that structure the basic content that populates the website. The majority of these are derived from `views/base.html`, which provides the base template for the entire site. From there, several blocks are defined, including a CSS block for adding additional stylesheets, a CONTENT block for adding content dynamically depending on the page, and a SCRIPTS block, which allows users to add specific JavaScript files for each different page type. This is especially important, as it is not necessary to add the layout creator functionality to every page, but just for the pages that require it.

A couple of *includes* are also defined within views, namely the header and footer of the website. Unlike `views/base.html` and those that inherit from it, the includes are files that are merely substituted into files that require them, but don't need inheritance based on the page being viewed. The header and the footer of the site are good examples for how includes work within our project, as they do not change between pages.

Public (public/)

The `public/` folder as it is, exists for the purpose of storing static files that will be served directly from the server. Page stylesheets, client-side JavaScript files, and images are all types of files that would ideally be stored in the public folder. Even static html files can be hosted within the `public/` directory, however, this is typically unwise for several reasons.

First and foremost, it's typically easier to create additional pages using templates, which will also

give you a consistent style across your page (since the template will take care of common stylistic traits). Furthermore, because of the way Express.js works, templates and routes defined in the `routes/getHandlers.js` file are rendered with priority over static files. In the case that a static file has the same name as one of the routes in `routes/getHandlers.js`, the application will return the corresponding route instead of the static file. In practice, you typically will want to define most of your content through `views/`, and use `public/` for stylesheets and client-side JavaScript files.

Models (models/)

Like with the views directory, the `models/` directory specifies the *models* component of our MVC framework. In particular, this directory only contains one file, which defines the model that defines our database. The specifics of the model itself are saved for another post, but if you want to extend our model beyond login information and store layout data, modifying or adding a schema to this folder would be the first place to start. Note that it may also be necessary to edit other areas of the code to accommodate new models, however this will vary depending on the scale of change you wish to make.

Configuration (config/)

Lastly, configuration for various components within the *controllers* is put in the `config/` folder. An example is the global configuration, which lies in `config.js`. In particular, we defined two configuration types: development and production. This way, our server can run in either mode, and change the configuration dynamically without having to change anything between the two servers.

`config/passportConfiguration.js` defines the structure for using local authentication with the Passport [29]. This is complementary to `config/auth.js`, which provides some helper functions for checking user authentication. While it seems strange to put the helper functions there, they were placed for convenience as they directly relate to Passport itself, and are typically used in conjunction when passport is used or called.

With the brief descriptions above, you should be able to find and change much of the code within the project. Moreover, the importance of splitting the project into separate components such as outlined above should be considered. In many cases, small changes to various components (such as in the templates within the `views/` folder) can result in large changes across the site. However, since this separation decouples much of the data and the logic of the application, we find that even if large changes occur, they seldom affect other pieces of the code. The most strongly coupled piece of code above is likely to be the model(s). This is an unfortunate necessity of an application that requires authentication, but it is manageable since extending the model is possible without affecting more than two or three different files at most (i.e. even if you extend the model, you won’t have to extend the existing routes and handlers, etc.).

Model Implementation

As mentioned above. one of the great things about using Node.js is the wealth of modules made available through the Node Package Manager. Thanks to this, it’s often very easy to find a module that will meet your requirements, and help you ship your final project much more rapidly than if you were to write each individual component yourself.

For the purposes of implementing the use cases which pertain to user authentication, we chose to use Passport.js [29] in combination with MongoDB (via the Mongoose database wrapper). The great part about all of the above is that while each individual module is great on their own, they can easily be swapped out with other databases / database managers if you have different requirements.

The actual implementation of our schema can be seen below. As can be seen, the code itself is rather

short, thanks to the expressive power that Mongoose provides as a database wrapper.

```
UserSchema = mongoose.Schema({
    username:  String,
    userData:  String,
    salt:      String,
    hash:      String
});
```

We can break the above schema down as follows:

- `username` is the name that the user registered with
- `salt` is the random data we use to form a cryptographic hash of the user's password. Wikipedia, while not terribly academic in nature, has a wonderful definition of the subject matter .
- `hash` is the hash of the user's password hashed with the corresponding cryptographic salt. For the purposes of generating this and the corresponding `salt` parameter used above, we opted to not implement our own cryptography, so we use the wonderful pwd [30] module available through NPM.
- `userData` is a string that defines a JSON object, of which constitutes the data that is saved from the Layout Creator Tool, discussed below

From this, we then defined two functions in particular, `signup` and `isValidUserPassword`, which were used to register new users, and validate user passwords respectively. While the code is openly available on Github [5], the following implementation for `isValidUserPassword` below has been provided as a reference:

```
UserSchema.statics.isValidUserPassword = function(username, password, done) {
    this.findOne({ username: username }, function(err, user) {
        if(err) {
            return done(err);
        }
        if(!user) {
            return done(null, false, { message: 'Incorrect username.' });
        }
        hash(password, user.salt, function(err, hash) {
            if(err) {
                return done(err);
            }
            if(hash === user.hash) {
                return done(null, user);
            }
            done(null, false, { message: 'Incorrect password' });
        });
    });
}
```

This function is then later put to use when we configure Passport.js, in the `config/passportConfiguration.js` file. Specifically, it's used in the exports in the following way:

```
module.exports = function(passport, config) {
    passport.serializeUser(function(user, done) {
        done(null, user.id);
    });

    passport.deserializeUser(function(id, done) {
        User.findOne({ _id: id }, function(err, user) {
            done(err, user);
        });
```

```
    });

    passport.use(new LocalStrategy({
        usernameField: 'username',
        passwordField: 'password'
    },
    function(username, password, done) {
        User.isValidUserPassword(username, password, done);
    }));
}
```

The above configuration relies on `isValidUserPassword` from our User model to validate if a user's password was entered correctly. Thus, the only significant coupling of our model with Passport occurs at this point, which means that outside of the model itself, you should only have to change this file dependency in order to use a different model with Passport.

Lastly, I should talk about how all of this is put to use within the server. The short snippets above show how Passport will serialize you as a user, and validate whether or not you entered your password correctly. However, the server itself uses the wrapper functions provided in `config/auth.js` to assist with checking user authentication in the route handlers. See the function definitions below:

```
module.exports = {
    isAuthenticated : function(req, res, next) {
        if(req.isAuthenticated()) {
            next();
        } else {
            res.redirect('/login');
        }
    },

    userExists : function(req, res, next) {
        User.count({
            username: req.body.username
        },
        function(err, count) {
            if(count === 0) {
                next();
            } else {
                res.redirect('/signup');
            }
        });
    }
}
```

This is taken from `config/auth.js`. In particular, by using these configuration wrappers, we can test for two things: If a user exists, and if the current session is authenticated with a specific user profile. For defining pages in `routes/getHandlers.js` that require authentication, or adding additional routes to `routes/postHandlers.js` for sending data in an authenticated session, you'll want ot use the `isAuthenticated` function as follows in the application:

```
app.get('/home', function(req, res) {
    Auth.isAuthenticated(req, res, function() {
        var templateParameters = {
            // Metadata options
            "title"       : "Home",
            "authors"     : authors,
            "description" : false,
            "user"        : req.user.username,
            // Navbar options
            "navStatic" : true,
            "home"     : true,
```

```
            };
            res.render('home.html', templateParameters);
        });
    });
```

Note that in this specific example, we first check for if a user is in an authenticated session, and then we render/output the template with the above parameters. Of interest is the `req.user.username` seen above, which returns the username of the current user who is logged in. You can choose any field from the schema defined at the beginning of this post, but I would warn against sending the salt or hashed password, as that could potentially destroy your user security entirely.

One interesting parameter that can be returned is the `req.user.id` parameter, which was not explicitly outlined in the schema above. This is a parameter that is automatically generated by Mongoose when the schema is compiled into a Model object, which happens at the end of `models/userSchema.js`. One notable use case for this is making GET and POST requests to specific fields for users. Take for example the `userData` field that each user has, which needs to be obtained and updated every time the store layout configuration changes:

```
app.get('/get-user-data', function(req, res) {
    Auth.isAuthenticated(req, res, function() {
        User.findById(req.user.id, function(err, doc) {
            console.log(doc.userData);
            if(err) {
                res.send(500);
            } else {
                res.set({
                    "Content-Type": "application/json",
                    "Content-Length": doc.userData.length
                });
                res.send(doc.userData);
            }
        });
    });
});
```

The above will return the JSON string stored in the `userData` field, using some built in functionality of our User model (returned by including `models/userSchema.js`) our authentication helpers (defined in `config/auth.js` above). If you want to learn more about these in detail, I highly suggest reading the extensive documentation and API reference available both for Passport (guide) and Mongoose (docs).

Use Cases

Use Cases Implemented

2.8 – Log In / Authenticate

A user can register for the website by visiting the signup page from within their browser. After registering a username and password, they can log in to the site and authenticate their session with the server. Afterwards, they are able to save and load user data that specifies the data format for viewing the store layout.

Use Cases Abandoned

2.9 – Become System Configurator

Due to time constraints, extending the user authentication model to allow for different tiers of users, as well as incorporating organization level grouping of users was not implemented. Ultimately, each user has their own individual profile and data, which may not be the expected behaviour of the final

application; however, while this may be the case, as a prototype of a final system, the functionality of the model that we specify below doesn't hinder the implementation or usage of any other use-case or feature.

From a technical standpoint, we have currently implemented this in such a way that any *Authenticated User* is automatically promoted to *System Configurator* status. However, while we realize that this is not the intuitive interpretation of the use cases, we chose not to pursue this avenue further in order to better implement more primary functionality of the site.

Client / Front End Development

Store Layout Creator

The Store Layout Creator page made up a significant portion of the front-end software development effort. There was a need to create a system by which the user can create a digital representation of a physical space. The format chosen was one that resembles inventory plans that are used in large stores to organize products. Efforts were made to build a system that can be altered dynamically in order to create an accurate and up to date model. In this section, we talk about the development details of the Store Layout Creator.

Using the Layout Creator

The first step on the web-side of the LASS system that any user would take would be to set up their store layout using the Store Layout Creator. This can be accomplished by clicking on the 'Create Layout' link on the navigation bar (depicted in Figure 18).

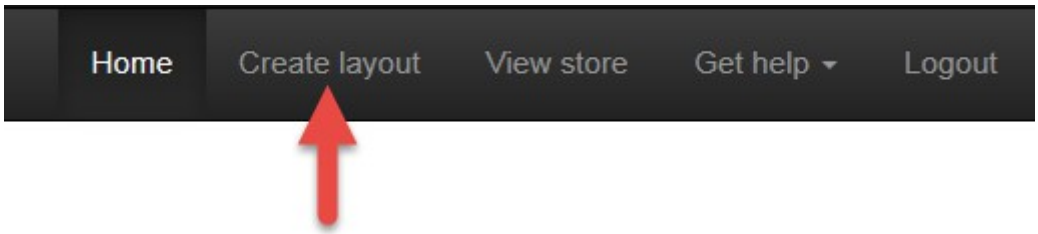

Figure 18: Screenshot of the navigation bar

Provided that the user is authenticated, this brings up the Store Layout Creator, which is shown below in Figure 19:

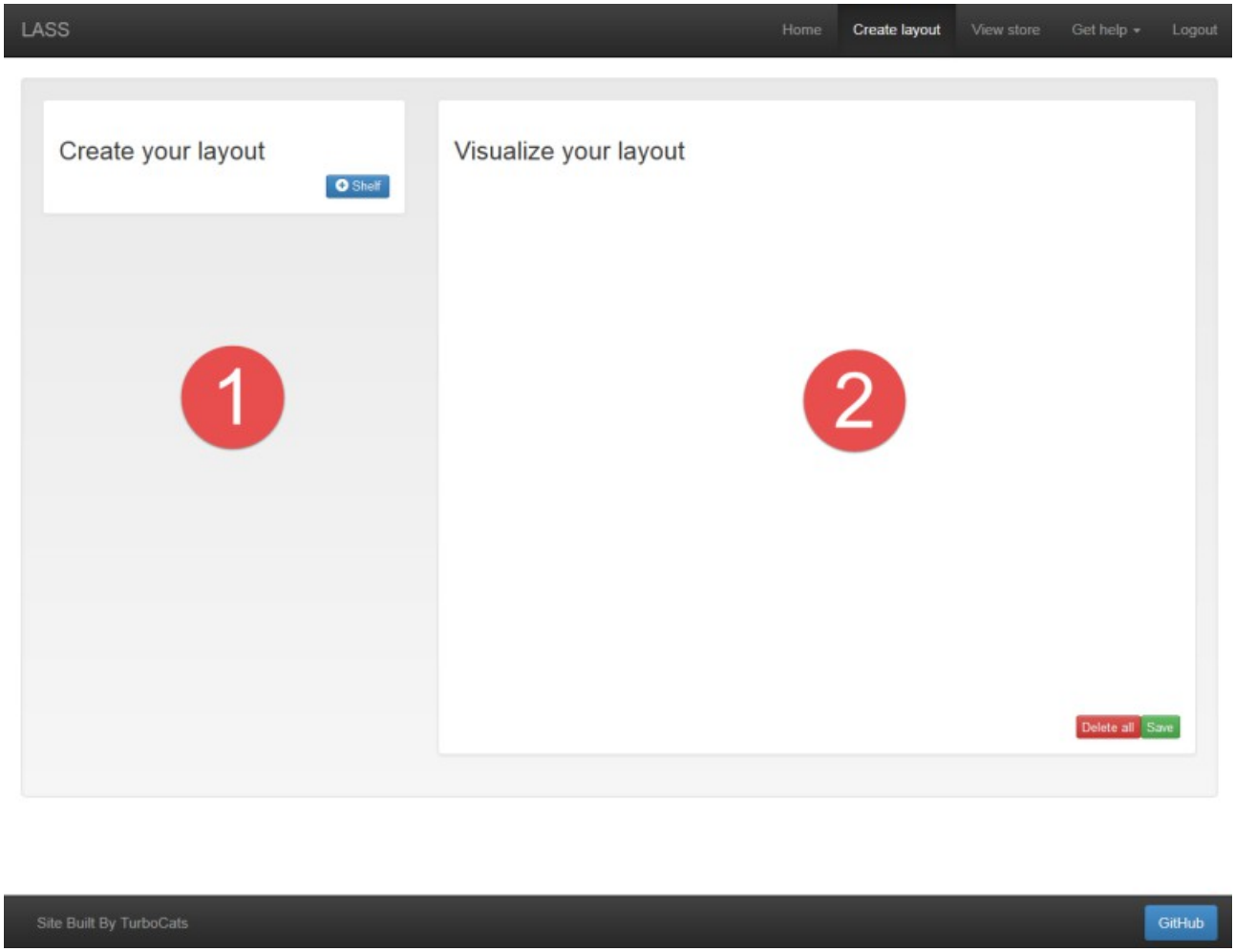

Figure 19: Initial page a user will see if they have an empty store layout (have yet to create one)

Notice that the user interface is split into two sections. The left section, labelled 1, is where you can configure the details of store layout.

The right section, labelled 2, is where you will see the visual representation of your store model created. This is updated in real time to reflect the changes you make to your layout.

To get started, click the '+ Shelf', which will add a shelf to your store, shown in Figure 20.

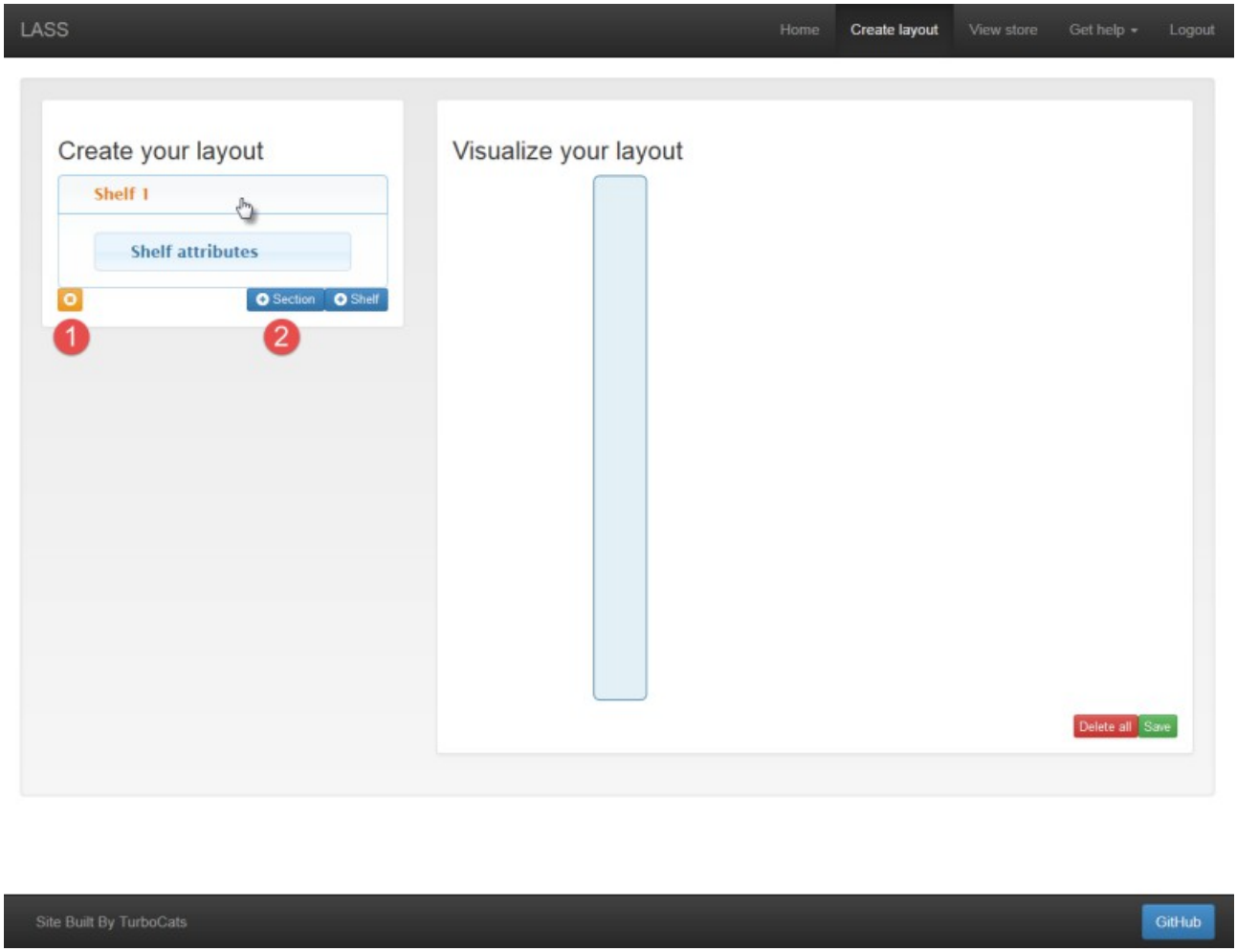

Figure 20: Depiction of when a shelf is added to the store layout

You will notice that a new panel, called an accordion, is added to the left section of the screen. This accordion can be opened and closed by a simple mouse-click. Accordions are used to keep the interface organized and accessible without the need for excessive scrolling. You will notice that the accordion contains a smaller accordion within it, called ‘Shelf Attributes’. This is automatically generated with each shelf, so that you can give some characteristics to the shelf as a whole.

Also within the left section, you will see that two new options have become available. The first, labelled 1, is the remove button. This allows you to remove whatever is currently selected from your store layout. In the screenshot above, clicking the remove button would remove ‘Shelf 1’ from your layout.

The second new option, labelled 2, is the ‘+ Section’ button. This is used to organize content on your shelf. For example you may have a shelf in your store dedicated to pet food. This may be split up into dog food, cat food, toys, and bird seed.

In addition to changes in the left panel, you will notice that a new shelf has appeared in the right section. It looks pretty bare-bones right now, but don’t worry, we are about to make things more interesting!

Lets go ahead and create some section for our pet food aisle by clicking the ‘+ Section’ button four times.

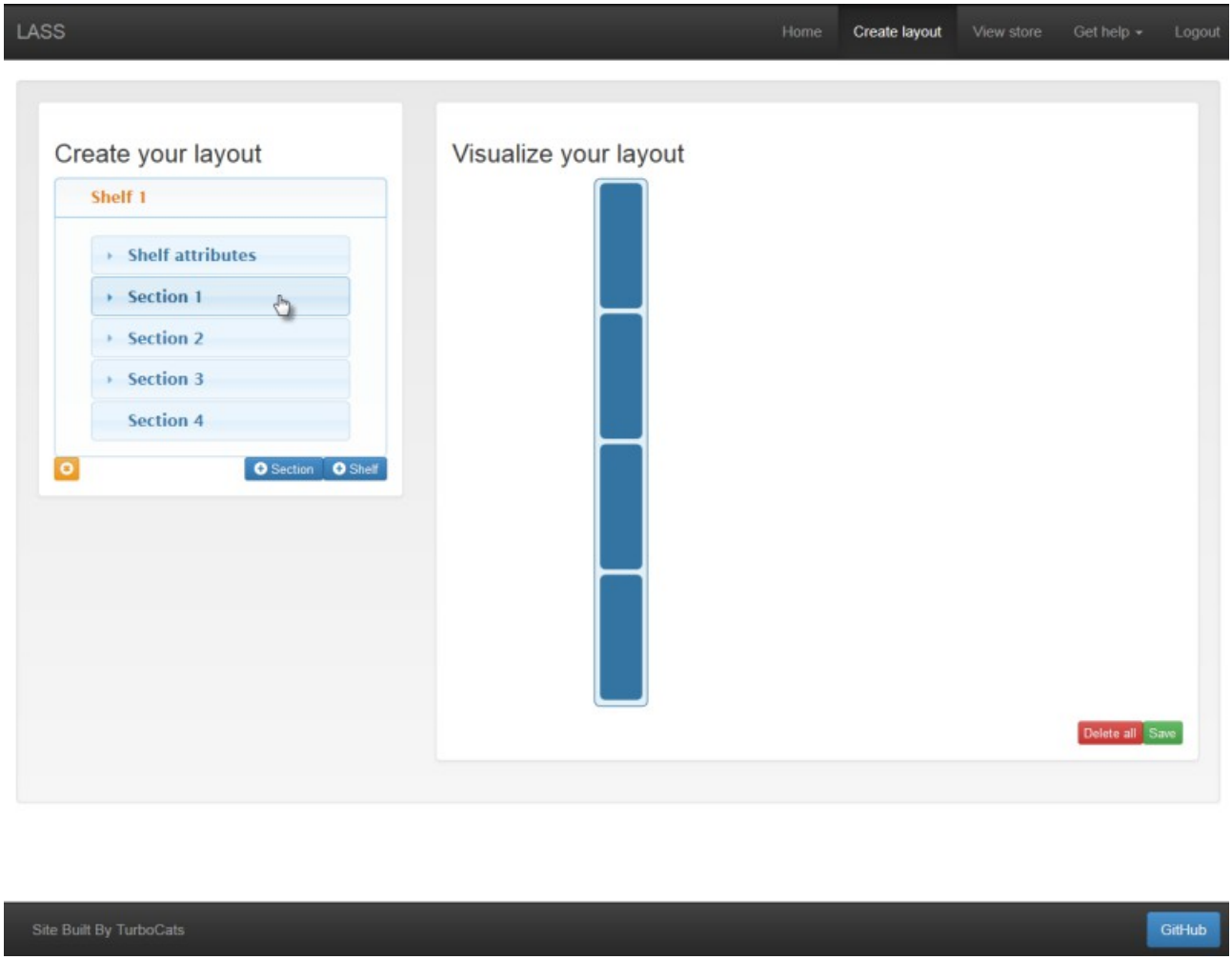

Figure 21: Depiction of what happens as you add sections to your shelves

As you have probably come to expect, section elements are added to the left section of the screen, and visualized on the right. From this point we can open on of the sections and add some details about it.

Although the right section is mostly for visualization, it also allows you to easily select jump to the details of a specific section of a shelf. If you click the element shown in Figure 21 above, Section 1 of Shelf 1 will open. Let's try that now. You should see what's shown in Figure 22.

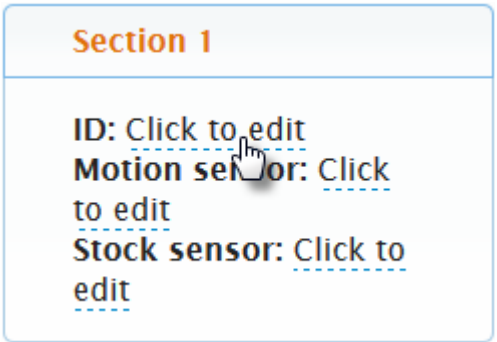

Figure 22: A single section with editable attributes

We want to create an aisle for all of our pet products. For now, let's give each of the sections an ID. Just click to enable editing, type in the desired name, and press enter to confirm. If you decide you don't want to change the text in this box, you can either click outside of the box or press ESC.

With our Dog Food, Cat Food, Toys, and Bird Seed sections created, we have a good start on creating a model for our store. By combining all of the concepts shown here, we can flesh this out to a few aisles shown.

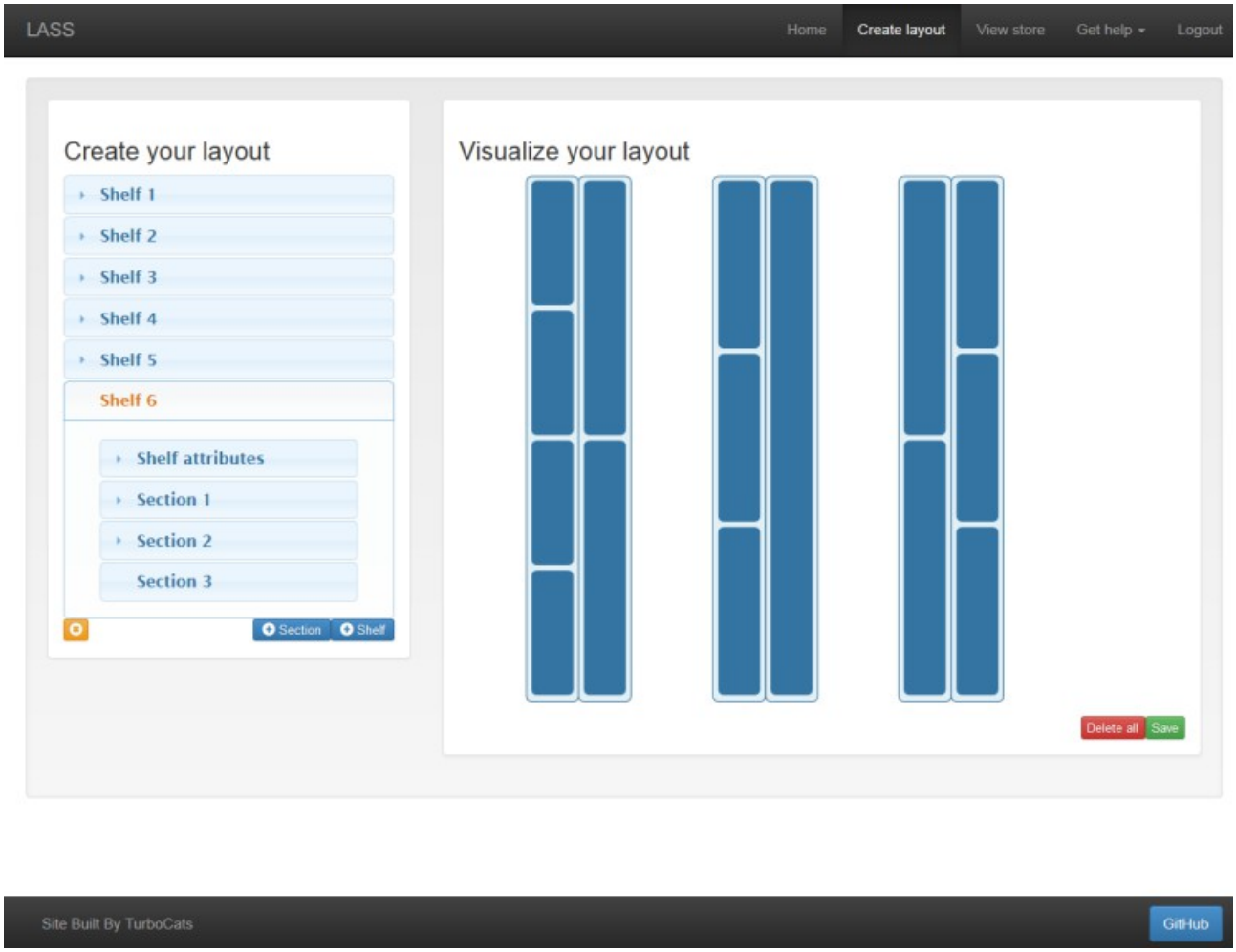

Figure 23: Complete Store Layout

Alright! We have a pretty nice looking store (Figure 23). When you are working in the Store Layout Creator, any changes you make are temporary until you choose to save them to the database. If you take a look at the bottom right hand side of the Store Layout, we can see buttons for ‘Save’ and ‘Delete All’. Simply click the ‘Save’ button and your layout will now be available to use and update as you see fit. If you make changes that you are not happy with, you can reload the page and you will be brought back to your last save state. If you want to scrap the whole thing, press the ‘Delete All’ button and start again!

The next step of using the system is to link the store sections to the sensors you have set up in your store. If you would like an explanation of how these systems work together, check out Using the Data Model.

Linking a sensor to a section is as easy as adding the URL of the datastream into the motion sensor text field. For example, we can add a motion sensor to the ‘Dog Food’ section as follows:

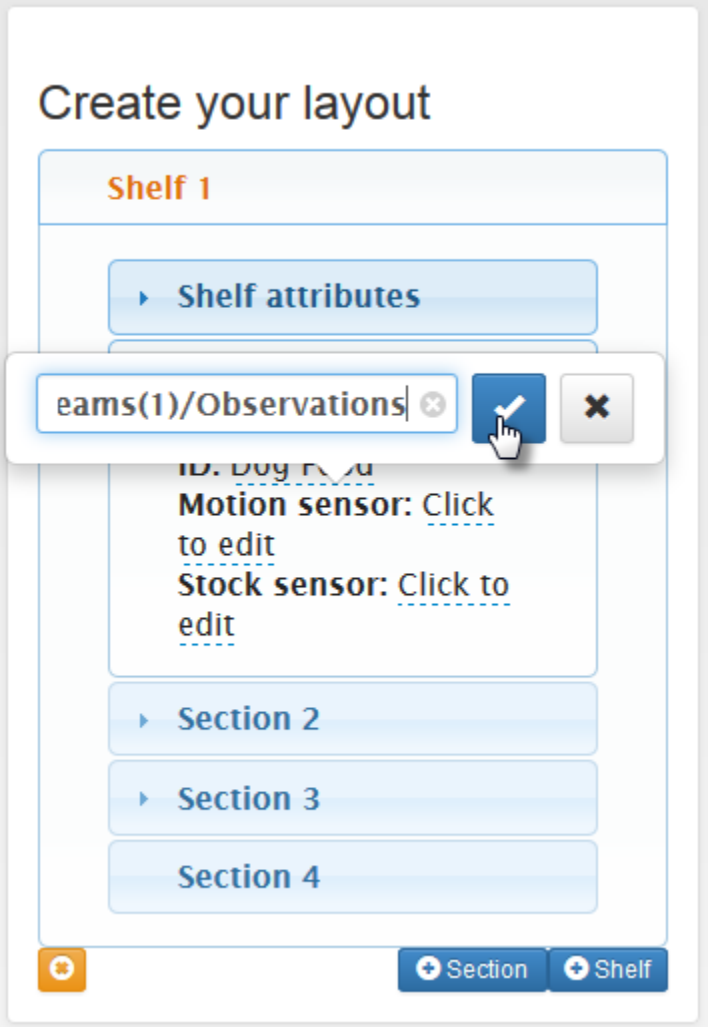

Figure 24: Linking Section to Data Model

Linking a stock sensor is done in the same way, just use the stock sensor text field instead. After adding all of your sensors, your store model will be ready to use. Don't forget to save your changes before exiting!

Enabling Technologies

To enable the user to build a model dynamically, it was decided that both a visual representation and easy to modify controls were needed. The user interface that was eventually decided upon was uses the Twitter Bootstrap front end framework [31] for positioning and styling, jQueryUI accordion elements [32] to organize the user's information, X-editable [33] to allow real-time editing of attributes, and D3.js [34] to visualize the store floorplan. To simplify the JavaScript necessary to merge all of these technologies together, jQuery [35] was utilized.

Store Layout Creator Architecture

Shown below in Figure 25 is a screenshot of the system with some shelves and sections added:

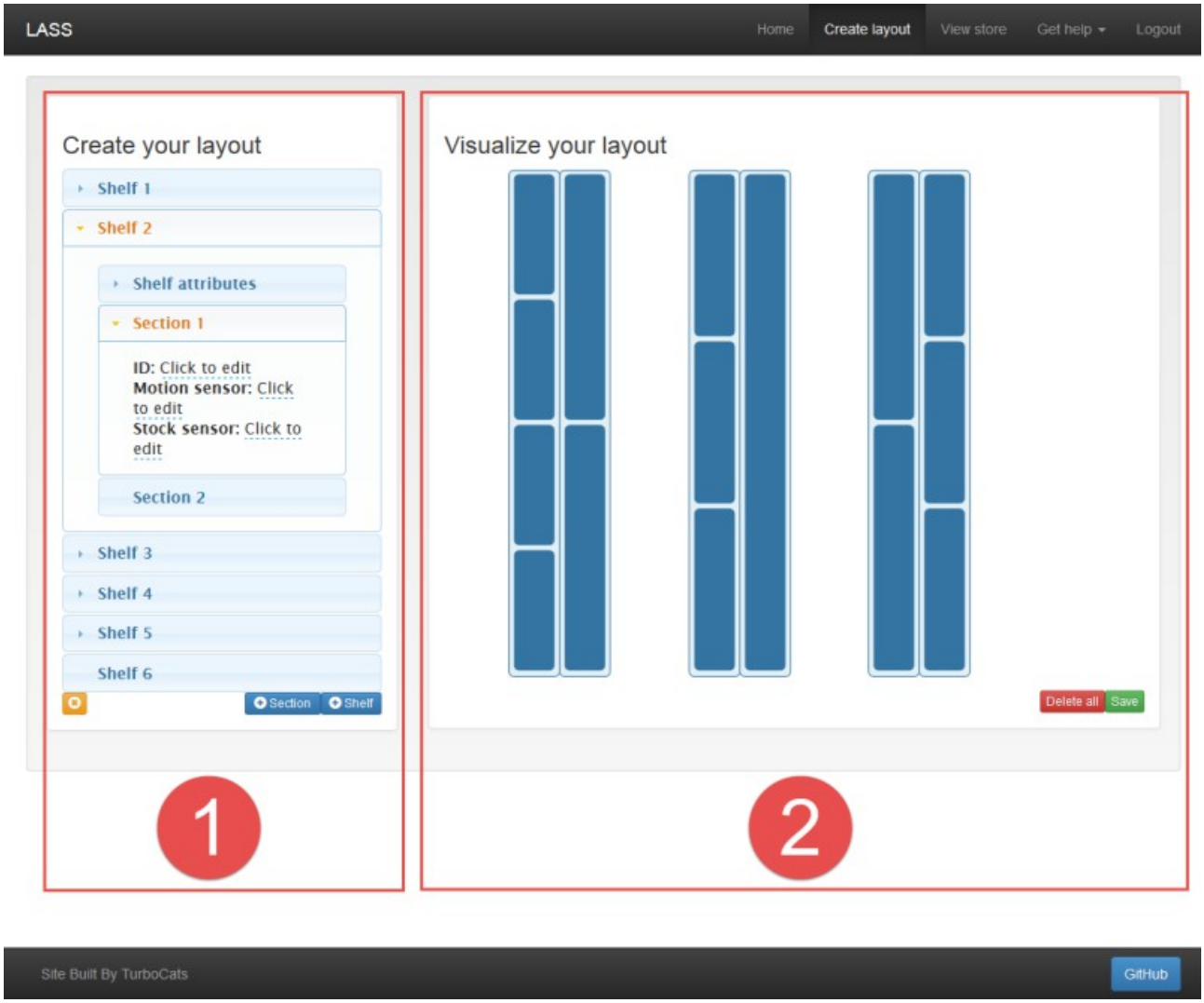

Figure 25: Screenshot of the layout creator in action

System Overview

As you can see in the screenshot above, the system is split into two distinct panels. When a user wants to create a model of their store, what they are really doing is manipulating a JavaScript object with the controls that have been provided to them. While both the left and right panels are representations of the same object, they present this data in a different way.

The user is given control over the JavaScript object via the controls of the left panel, labelled 1. The elements labelled ‘Shelf 1’, ‘Shelf 2’, etc., are jQueryUI accordion elements. Note that inside of the open element, ‘Shelf 2’, there are a set of nested accordions. These contain the attributes and sections of ‘Shelf 2’. By utilizing the ‘+ Section’, ‘+ Shelf’ and ‘(x)’ (remove) buttons, users can add to the object or remove the selected accordion element. Within each of the nested accordions, editable text fields exist. These allow users to configure the attributes of each part of their model.

The visual representation of this model is shown in the right panel, labelled 2. This is displayed to guide the user’s creation by allowing them to relate what they are building to physical space. The visualization is updated in real time as elements are added and removed. In the screenshot provided, the shelves are represented by the 6 light blue rectangles that contain an assortment of smaller blue rectangles. They are created as from left to right, and are in groups of two to represent two opposing sides of a shelf. Thus, the white space between the shelves represents the concept of an ‘aisle’. The smaller blue rectangles contained within them represent Sections, which can be used to set up

logical divisions to differentiate products on a shelf. They are created from top to bottom.

Once the user is happy with the layout of their store, the JavaScript object is converted to JSON and saved to database. Anytime the Store Layout Creator is opened thereafter, the object is loaded and the accordion elements, attributes, and visualization are rendered exactly as they were configured when the user saved. This allows the user to edit their store layout at any time without needing to build a new one from scratch.

Data Structure

An example of the object that is being manipulated and displayed by the user interface is shown below:

```
shelves = [
    {
        "sections": [
            {
                "displayID":"Dog Food",
                "pirURL":".../Things(1)/Datastreams(1)/Observations",
                "pintURL":".../Things(1)/Datastreams(2)/Observations"
            },
            {
                "displayID":"Cat Food",
                "pirURL":".../Things(2)/Datastreams(3)/Observations",
                "pintURL":".../Things(2)/Datastreams(4)/Observations"
            },
            {
                "displayID":"Bird Seed",
                "pirURL":".../Things(3)/Datastreams(5)/Observations",
                "pintURL":".../Things(3)/Datastreams(6)/Observations"
            }
        ],
        "notes":"Pet Food Shelf"
    },
    {
        "sections": [
            {
                ...
            }
        ]
        "notes":"Some other Shelf"
    }
]
```

The `shelves` array holds the `shelf` objects which, in this example, are comprised of `notes` which is assigned in the ‘Shelf Attributes’ accordion, and an internal array for `sections` which holds the `section` objects. There are three sections in the first shelf, for ‘Dog Food’, ‘Cat Food’ and ‘Bird Seed’. Alongside these attributes is the `pirURL` and the `pintURL`. These are the addresses of the observations of the sensors that are contained within those sections. A second shelf exists, with some filler text for its attributes.

Technical Challenges With Layout Creator

This section details some of the challenges encountered in building the Store Layout Creator. It serves to explain the inner workings of the system to guide anyone that wishes to use or modify our code.

Nested Accordions

jQueryUI accordions are based off of pre-defined html structures which are converted to an accordion once the accordion events are attached to them. In order to facilitate this in an on-the-fly

manner, the first step was to create a template of an accordion and house it in the HTML document for the page. We also need a ‘parent’ accordion for all of the new accordion elements to be added to. These can be seen below:

```
<!-- Parent for all accordions -->
<div id='parentAccordion'></div>

<!-- Template for accordions -->
<div class='accordion'>
    <h3 class='accordion-title'></h3>
    <div class='accordion-content'></div>
</div>
```

Then, the accordion events must be attached to `#parentAccordion`. The options available are defined in the documentation.

```
var accordionConfig = { ...options... };
var $parentAccordion = $("#parentAccordion");
$parentAccordion.accordion(accordionConfig);
```

With our accordion template defined, we can clone each time a new accordion is needed.

```
var $newAccordion = $('.accordion').children().clone();
```

Each of these clones will require the accordion events to be attached to them, just like `#parentAccordion`.

```
$newAccordion.accordion(accordionConfig);
```

Then we append them to `#parentAccordion`.

```
$parentAccordion.append($newAccordion);
$parentAccordion.accordion("refresh");
```

The accordion elements are created with the following:

- The header (title bar) with `id="#ui-accordion-parentAccordion-header-" + n`
- The panel (body) with `id="#ui-accordion-parentAccordion-panel-" + n`

Where n is the nth element in the accordion.

Utilizing these selectors, you can change the title and body contents. Additionally, by selecting the body of an accordion, you can attach a nested accordion by following the steps described above.

Removing Shelves and Sections

In order to remove the active shelf or section, functionality to remove elements from the accordion as well as the d3.js visualization was needed. It is trivial to remove an element from the `shelves` array using `.splice(n,1)` but some additional work must be done to update the accordion and d3 SVG elements.

Removing accordion elements

Since we extended the abilities of the jQueryUI accordion to allow it to be used in a dynamic way, it follows that it does not have built in functionality to remove elements from a dynamic accordion. Problems arise when removing elements from the accordion because of the way they are assigned `ids`. It becomes necessary to reassign existing `ids` once elements are removed to re-instate a 0-n

order without gaps.

For example, if `shelves` array has 5 elements, and you remove element 3, the remaining elements would have headers labelled `id="#ui-accordion-parentAccordion-header-" + n`, where `n = 0,1,3,4`. Then, because of the gap in numbering, trying to add a new element will cause the accordion to behave undesirably.

The approach that we took to do enable deleting for the accordion element is as follows:

First we remove the accordion header and panel from the accordion.

```
var header = "ui-accordion-parentAccordion-header-";
var panel = "ui-accordion-parentAccordion-panel-";

$("#" + panel + activeSectionNumber).remove();
$("#" + header + activeSectionNumber).remove();
```

Then we rename the `ids` of the remaining accordion elements.

```
$("#parentAccordion > h3").each(function (i) {
    $(this).attr("id", header + i);
    var n = i + 1;
    $(this).text("Shelf " + n);
});
$("#parentAccordion > div").each(function (i) {
    $(this).attr("id", panel + i);
});
```

Then we must also dive one level deeper and rename the `ids` of any child accordion elements.

```
for( var i = 0; i < shelves.length; i++){
    var panelSelect = "#ui-accordion-parentAccordion-panel-" + i;
    var childHeader = "ui-accordion-ui-accordion-parentAccordion-panel-" + i + "-
header-";
    var childPanel = "ui-accordion-ui-accordion-parentAccordion-panel-" + i + "-
panel-";
    $(panelSelect + " h3").each(function (j) {
        $(this).attr("id", childHeader + j);
    });
    $(panelSelect + " div").each(function (k) {
        $(this).attr("id", childPanel + k);
    });
}
```

Removing SVG elements

While d3 does allow for the removal of elements dynamically using the `.exit()` function, the perceived gain in retaining all of the relationships between the visualization and the `shelves` array was low compared to the work required. Due to time constraints, it was decided that when an element is removed, that the visualization is redrawn similar to what happens on the initial loading of an existing configuration of the system. Being as the user's focus will likely be on the accordion elements during removal, this undesirable characteristic was decided to be okay for the first version of LASS.

Saving and loading editable attributes

Enabling editable attributes was done using the X-editable JavaScript plugin. This allows the text the user enters to modify the `shelves` array. Each time a new field is added to an accordion panel, it needs to be made editable. This is done by checking which new fields exist and adding the editable

events to them. An example for a field of class `motion` is shown below:

```
$(".motion").not(".editable").editable({
    ... options ...
});
```

The options that need to be defined to allow the manipulation of the `shelves` object are as follows:

```
defaultValue : pirURL,
success : function(response, newValue){
    pirURL = newValue;
},
display: function(value){
    if ( pirURL == undefined ){
        $(this).text("Click to edit");
    } else {
        $(this).text(pirURL);
    }
}
```

Where:

- `defaultValue` is what appears in the editable box
- `success` is what to do when the user enters a value
- `display` is what to display when the editable field is created

D3.JS

Apart from the seemingly steep learning curve of d3.js, there were no additional factors that added to the difficulty of creating the visualization tab. All of the functionality implemented works in a standard way.

Store Viewer

Enabling Technologies

Similar to the Store Layout Creator, the Store Viewer relies on a few different technologies. First, the user interface uses the Twitter Bootstrap front end framework for positioning and styling. Also D3.js is used to visualize the store, observations, and statistics. To simplify GET requests and the JavaScript necessary to merge all of these technologies together, jQuery was utilized.

Store Viewer Architecture

Shown in Figure 26 on the next page is a screenshot of the system with some active observations being displayed:

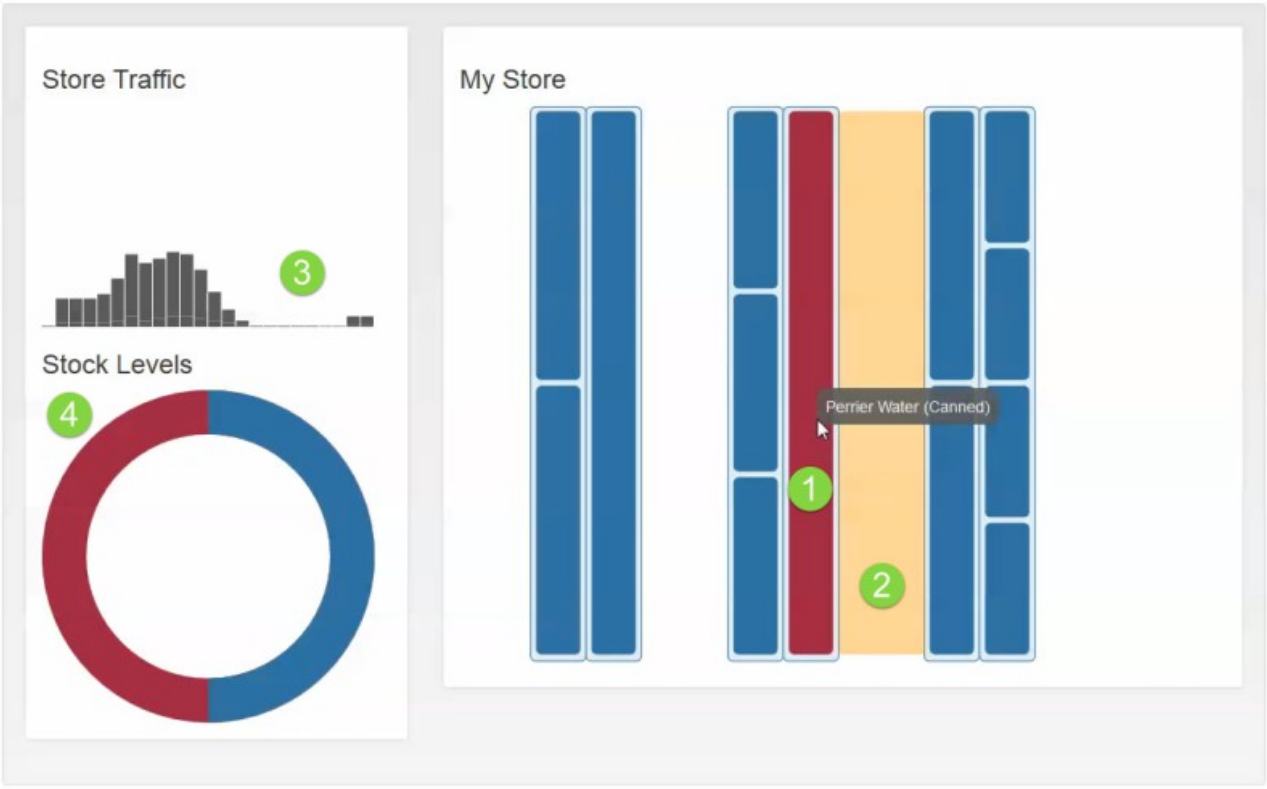

Figure 26: Screenshot of system working with active observations being displayed

System Overview

As you can see in the screenshot above, there are 4 different types observation representations being displayed. The left pane shows overall statistics, and the right pane shows a real-time representation of the store.

For real time-stats, there are two ways to display observations. The first type of observation that is visualized is an empty or full section. Full sections appear blue, and empty sections appear purple, as shown by label 1. The second type of observation that is visualized is traffic within an aisle. This is represented by a ‘heat map’. When someone walks past a sensor, an orange section is displayed, as shown by label 2.

For the overall statistics, there are also two ways of displaying information. The first is an overall level of traffic within the store, shown by label 3. This takes the number of readings from all motion sensors for an epoch in time and displays them in a bar graph that updates as new epochs become available. The second is an overall stock level, shown by label 4. It calculates the overall stock level by taking the number of full and empty sections for each epoch.

Other Features

Another feature that couldn't be seen above is the ability to view the history of recent observations in the viewer. See Figure 27 below to show an example of the history viewer:

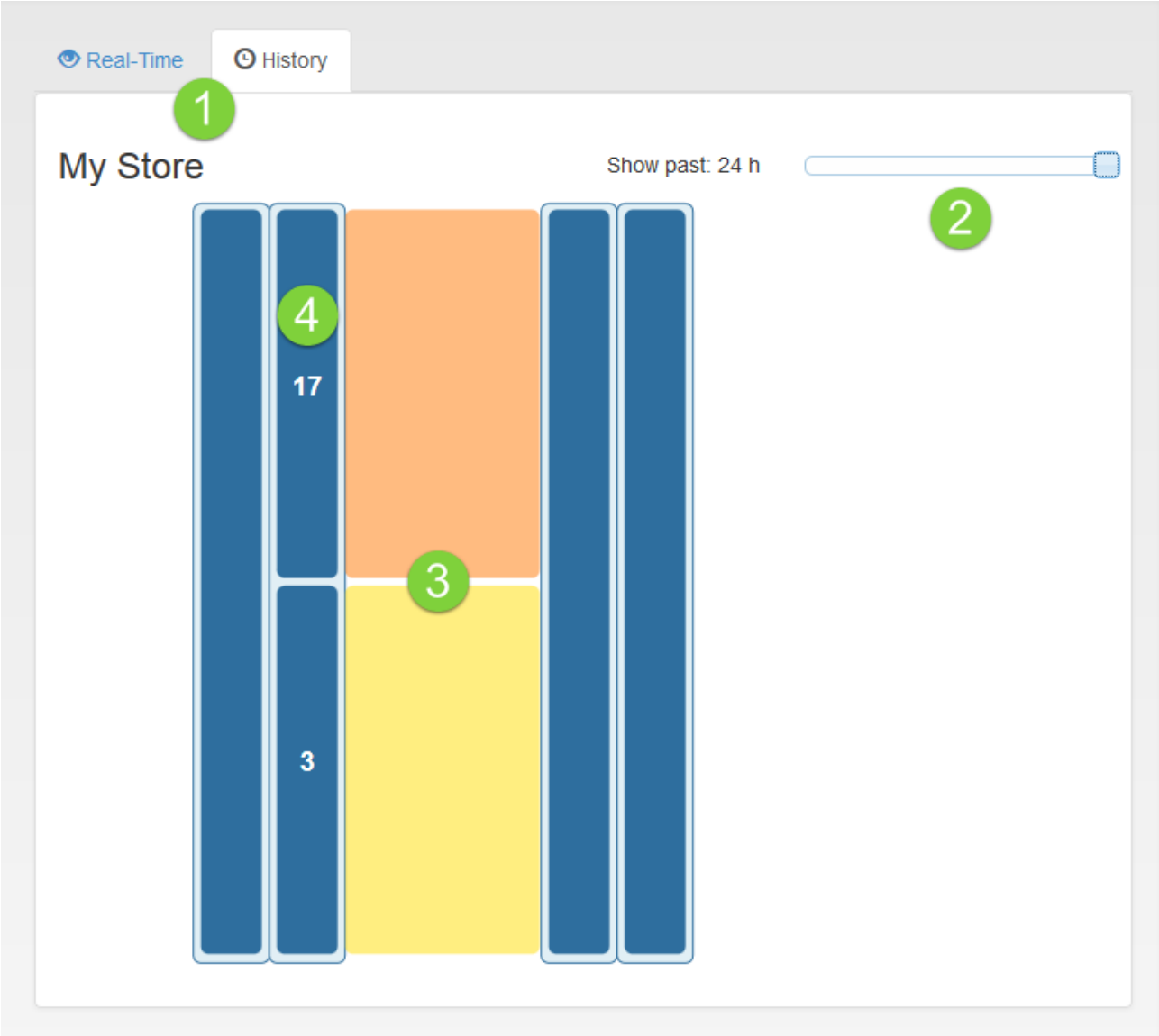

Figure 27: Example of viewing historical data for your store

At the very top, you can see there are two new tabs added in: Real-Time and History (marked by 1 in the figure). Both modes will render the same store layout, but different observations. The default tab, Real-Time, shows the current traffic and stock information as our original version did. However, if you select the History tab instead, you can see traffic and stock information based on the past 1-24 hours.

In the top right-hand corner, there is a slider (marked by 2 in the figure). This slider is used to select the number of hours you would like to view. For example, if you want to see aggregate traffic and stock data for the last 14 hours, you can slide it to 14. In the screenshot shown, we selected 24 hours.

Once you've selected a time frame, the traffic and stock observations will be overlaid on the store map. The heat map (marked by 3 in the figure) is rendered much like the real-time version, except this time the colour of the heat map corresponds to the total number of motion sensor observations (i.e. how many people passed by that section in the past, for example, 24 hours). Yellow corresponds to fewer traffic observations, and the colour varies from yellow to orange to red as the number of traffic observations increases.

Finally, stock observations are also updated (marked by 4 in the figure). The number in each section is reflective of the number of times that section was emptied in the past time period. In our example, one of the shelves had been emptied 17 times in the past 24 hours, whereas the shelf adjacent had only been emptied 3 times in the past 24 hours.

Future Work

While this feature is really neat and we think it adds a lot of value to the store owner, there are still some small user experience items to be added in the future. We added a colour bar/legend so that it's more clear what heat map shades correspond to in terms of real numbers, but we have not been able to test the device in a real store. Because of that, the scale values are somewhat arbitrary and would need to be adjusted for realistic in-store usage.

Technical Challenges With Layout Viewer

This section details some of the challenges encountered in building the Store Viewer. It serves to explain the inner workings of the system to guide anyone that wishes to use or modify our code.

Getting Observations with GET Requests

Making GET requests, in general, is easier than making POST requests. We are essentially just reading from the server what information is there, so all we really need is the correct URI to read from. In our case, we are interested in retrieving observations, which can be easily read from a URL provided you have the following information:

- Thing ID (e.g. 7)
- Datastream ID (e.g. 12)

You'd construct the URL – based on our example values – as follows:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things(7)/Datastreams(12)/Ob
servations
```

(Don't forget to change your root URI as appropriate!)

Unfortunately, after we've been running our Raspberry Pi for a few hours, we are going to end up with potentially thousands of observations. If we make a GET request to that URL, we're going to end up with a huge object, and that's just no good.

However, the data service has functionality for filtering – you can read up on it more in the API reference, under section 2.2.4 [1].

For our purposes, we used two of the possible filter capabilities. The first one was to filter ResultValue, so that we could get only Observations with a ResultValue property of 1. (This made it easier to render the store traffic statistics, as discussed in another sectopm.) To do this, you make a GET request to the following URL:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Things(7)/Datastreams
(12)/Observations?$filter= ResultValue eq '1'
```

The second filtering capability we used was filtering by time. For example, if you want observations after 12:00 noon on April 4, 2014, you would use the following URL:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0/Observations?
$filter=Time ge STR_TO_DATE('2014-04-04t12:00:00-0600','%Y-%m-%dt%H:%i:%s')
```

The "ge" stands for greater than or equal to and "eq" from the previous URL stands for equal to (as

you may have guessed). You can also combined multiple filters, simply by adding “and” to your URL:

```
http://demo.student.geocens.ca:8080/SensorThings_V1.0_students/Observations
?$filter=Time ge STR_TO_DATE('2014-04-04t12:00:00-0600','%Y-%m-%dt%H:%i:
%s') and Time le STR_TO_DATE('2014-04-05t12:00:00-0600','%Y-%m-%dt%H:%i:
%s')
```

This request would get all observations between 12:00 noon on April 4 and 12:00 noon on April 5. We used these two types of filters in order to update our observations without pulling ridiculously large objects.

Getting Observations Asynchronously

One of the challenges we ran into was managing the many GET requests being made from the website. When dealing with multiple sections, we were making requests for each individual section and the response latency sometimes shifted the way that observations were rendered. To deal with this at first, we set all of our requests to be done synchronously; however, we soon found that this created significant lag in response to simple things like navigating to a different page.

To make the asynchronous requests work, we created a separate function containing the GET request itself, which would also take in arguments for the shelf and section indices. This took care of the latency issue. (Sometimes, the response for one section would be drawn onto a different section because of the time lag). Here is an example of that code:

```
function doGet(shelfInd, sectionInd, URL, type) {
    jQuery.get(URL, function ( data, textStatus, xhr ) {
        if(xhr.status < 400){
            shelves[shelfInd].sections[sectionInd].obs = data;
            //Call function to render observations
        }
    });
}

// "Main" script
for( var i = 0; i < shelves.length; i++ )   {
    for( var j = 0; j < shelves[i].sections.length; j++){
        // Set the url depending on what type of observation it is
        if (obsType == "motion"){ // PIR Motion sensor
            var obsURL = shelves[i].sections[j].pirURL;
        } else if (obsType == "stock"){ // Photo interrupter
            var obsURL = shelves[i].sections[j].pintURL;
        }
        if( obsURL != null ){
        // Pass the variables of the get to a function so that indices don't get
borked
            doGet(i,j, obsURL, obsType);
        }
    }
}
```

Displaying Overall Statistics

The overall statistics, being a more standard use of D3.js, were completed by referencing existing tutorials. The store traffic display is a product of examining a dynamic stacked bar chart example, while the stock levels comes from this animated donut example [36].

Displaying Real-Time Observations

Once the observations become available, displaying the observations with d3.js is straightforward.

Changing the color for an empty or full shelf is just a matter of changing the fill color of the SVG. The heat map works by creating a 0% opacity rectangle SVG element that is positioned relative to the section that contains the motion sensor. If the motion sensor detects motion, the opacity is increased and then decreases slowly back to 0% using a timer.

Use Cases

Unlike with the webserver implementation, all of the remaining use cases as outlined in Figure 3 were completed. How each component above contributed to each use case is listed below:

Store Layout Creator

2.1 - Assign location to a microcontroller

This use case was defined as the need to assign a location to a microcontroller, but ended up evolving to the ability to assign a location to a sensor.

Assigning a location to a sensor is done by adding the url of a sensor's datastream to the editable text feilds located within a sections attributes. Too see how to do this, follow the instructions for *Using LASS: Store Layout Creator* in the section above.

2.2 - Configure a store layout

This is the main use case that is accomplished by the Store Layout Creator. Details of its workings are explained in the sections that follow.

2.3 - Show map UI

The map UI that is used for both the Store Layout Creator and the Store Viewer is created in real time while editing the store layout.

Store Layout Viewer

2.3 - Get aisle traffic observations

Observations from the PIR motion sensors are requested from the database. The URLs for these GET requests are taken from the `shelves` array configured in the Store Layout Creator.

2.4 - Get facing observations

Observations from the Photo-Interrupter sensors are requested from the database. The URLs for these GET requests are taken from the `shelves` array configured in the Store Layout Creator.

2.5 - Get aisle traffic

Using the observations obtained from use case 2.3, aisle traffic is displayed in two ways. The first is a heat map to show the location of customers within a store, and the second is a stacked bar chart showing overall traffic levels over time.

2.6 - See if products are faced

Using the observations obtained from use case 2.4, stock levels are displayed in two ways. The first is by showing which sections are empty, and the second is showing an overall stock level for the store.

2.7 - Show map UI

The data acquired from fulfilling use cases 2.4-2.6 are displayed in one cohesive user interface, the Store Viewer page.

Conclusion

Shortcomings of Project

As with any project of large scale, trade-offs had to be made to some degree in order for us to complete the important components in time. Across the entirety of the project, there were three major shortfalls that we regrettably could not complete due to time or cost restraints:

1. Our final Use Case, 2.9 – Making a user a store configurator, was not implemented completely.
2. We did not manage to find funding or outside support early on in the project, so while our sensor setup does work, we do not have a "shelf" to serve as an example as to how our hardware would be set up in a real world situation.
3. We did not manage to perform usability tests or do extensive software testing as we had hoped. (This is more of a user experience issue, not a technical bug issue)

When considering the shortcomings that are listed above, we had to ask ourselves one question – *do these shortcomings prevent us from meeting the primary objectives as listed in our project proposal?* In effect, while we had to fore-go extensive testing, we managed to get all of the primary functionality working in its base form. What's more, although we did not manage to fully implement use-case 2.9, we were still able to implement and showcase the remaining functionality of the system. Lastly, despite not having a physical shelf with which to demo, we can still show off the technical components that comprise the IoT-based components that we were after.

Ultimately, we felt that our objectives were met, and that LASS as it currently stands serves as a strong example of not only how to use the OGC SensorThings API, but how to develop an IoT-based application from start to finish. What's more, with the collaborative and open nature of our project, and given that it exists immortally on Github [23], it should continue to not only provide a base for others to start using and developing using the IoT and SensorThings API, but should also provide value in the sense that anybody can take our project in whole or in parts and use it in the real world.

Lessons Learned

Of particular importance is that it should be noted that many of the modules, components, and frameworks we took for granted in this project were not entirely without difficulty to learn. Comparatively, our group spent quite a bit of time in the planning and research phase of the project simply because there was a much higher learning curve to teach ourselves each of these technologies. That said, a considerable amount of software planning and engineering took place in order to organize the project enough to know where to start tackling it.

Therein lies one of the most important lessons of our project, which was project management and software design. While Geomatics Engineering does have a strong presence in software, our group found that it felt inadequate at times. To be more descriptive, web-based projects require a lot more thought and effort with regards to data concurrency, program scheduling, and interoperability. To bring this project to fruition, consideration for even some of the smallest variables had to be taken wholly into account, and it is for that reason that we feel that software design and project management were incredibly important lessons that we had to grasp in order to succeed.

Summary

Overall, the group feels that we have completed the project objectives and course requirements to the best of our ability, and we likewise feel that in the future we will succeed if we continue with

the skills we learned while trying to manage each separate piece both individually, and as a whole project.

Acknowledgments

We would like to thank Dr. Steve Liang, our supervisor, for his continued support both academically and otherwise throughout the project, and would like to thank the Department of Geomatics Engineering for assisting us with funding, meeting rooms, and other accommodations in order to make this project a possibility.

References

[1] "OGC SensorThings API," *OGC SensorThings API*, 2013. [Online]. Available: http://ogc-iot.github.io/ogc-iot-api/. [Accessed: 05-Dec-2013].

[2] "Here's Why 'The Internet of Things' Will Be Huge, And Drive Tremendous Value for People And Businesses," *Growth In The Internet of Things - Business Insider*, 22-Nov-2013. [Online]. Available: http://www.businessinsider.com/growth-in-the-internet-of-things-2013-10. [Accessed: 05-Dec-2013].

[3] Phillips Corporation, "Meet hue | en-US," *Meet hue | en-US*. [Online]. Available: http://meethue.com/. [Accessed: 05-Apr-2014].

[4] Belkin Corporation, "WeMo Home Automation," *WeMo Home Automation*. [Online]. Available: http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation/. [Accessed: 05-Apr-2014].

[5] Jeremy Steward, "ThatGeoGuy/ENGO500-Webserver," *ThatGeoGuy/ENGO500-Webserver*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500-Webserver. [Accessed: 13-Feb-2014].

[6] Kathleen Ang, Ben Trodd, Jeremy Steward, Alexandra Cummins, and Harshini Nanduri, "ENGO 500: GIS & Land Tenure #2 - Project Proposal," University of Calgary, Proposal 1, Sep. 2013.

[7] "What is RFID? - RFID Journal," *What is RFID? - RFID Journal*. [Online]. Available: http://www.rfidjournal.com/articles/view?1339. [Accessed: 05-Apr-2014].

[8] Chien Ko, "3D-Web-GIS RFID Location Sensing System for Construction Objects," *Sci. World J.*, vol. 2013, pp. 1–8, Jun. 2013.

[9] Carla R. Medeiros, Jorge R. Costa, and Carlos A. Fernandes, "RFID Smart Shelf With Confined Detection Volume at UHF," in *RFID Smart Shelf With Confined Detection Volume at UHF*, 2008, vol. 7, pp. 773–776.

[10] Raspberry Pi, "Raspberry Pi | An ARM GNU/Linux box for $25. Take a byte!," *RaspberryPi.org*, 2013. [Online]. Available: http://www.raspberrypi.org/. [Accessed: 17-Nov-2013].

[11] Arduino, "Arduino," *Arduino*. [Online]. Available: http://arduino.cc/. [Accessed: 10-Nov-2013].

[12] Netduino, "Netduino Hardware," *Netduino Hardware*. [Online]. Available: http://netduino.com/hardware/. [Accessed: 10-Nov-2013].

[13] Kathleen Ang, Ben Trodd, Jeremy Steward, Alexandra Cummins, and Harshini Nanduri, "ENGO 500: GIS & Land Tenure #2 - Literature Review: Location Aware Smart Shelves," University of Calgary, Literature Review 2, Nov. 2013.

[14] Jeremy Steward, Alexandra Cummins, Kathleen Ang, Ben Trodd, and Harshini Nanduri, "ENGO 500: GIS & Land Tenure #2 - Technical Deliverables Report," University of Calgary, Technical Deliverables Report, Dec. 2013.

[15] Ben Trodd, Alexandra Cummins, Jeremy Steward, Kathleen Ang, and Harshini Nanduri, "Use Cases - ThatGeoGuy/ENGO500 Wiki," *Use Cases - ThatGeoGuy/ENGO500 Wiki*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500/wiki/Use-Cases. [Accessed: 13-Feb-2014].

[16] "PIR Motion Sensor," *SK Pang Electronics, Arduino, Sparkfun, GPS, GSM*. [Online]. Available: www.skpang.co.uk/catalog/index.php. [Accessed: 05-Dec-2013].

[17] Hubert, "Shelf Management Solutions and Shelf Facing Systems," *Shelf Management Solutions and Shelf Facing Systems*, 2013. [Online]. Available: http://www.hubert.com/Display-Cases-Fixtures-0304/Shelf-Management-03040357.html. [Accessed: 02-Nov-2013].

[18] "Photo Interrupter," *SK Pang Electronics, Arduino, Sparkfun, GPS, GSM*. [Online]. Available:

https://www.sparkfun.com/products/9299. [Accessed: 05-Apr-2014].
[19] Jeremy Steward, Alexandra Cummins, Kathleen Ang, Ben Trodd, and Harshini Nanduri, “ENGO 500: GIS & Land Tenure #2 - Progress Report,” University of Calgary, Progress Report, Feb. 2014.
[20] Sparkfun Electronics Co., “PIR Motion Sensor Datasheet.” .
[21] “Hurl.it - Make HTTP requests,” *Hurl.it - Make HTTP requests*. [Online]. Available: http://www.hurl.it/. [Accessed: 05-Apr-2014].
[22] “Requests: HTTP for Humans,” *Requests: HTTP for Humans*. [Online]. Available: http://docs.python-requests.org/en/latest/. [Accessed: 05-Apr-2014].
[23] Jeremy Steward, “ThatGeoGuy/ENGO500,” *ThatGeoGuy/ENGO500*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500. [Accessed: 05-Dec-2013].
[24] “node.js,” *node.js*. [Online]. Available: http://nodejs.org. [Accessed: 13-Feb-2014].
[25] “Express - node.js web application framework,” *Express - node.js web application framework*. [Online]. Available: http://expressjs.com/. [Accessed: 13-Feb-2014].
[26] “MongoDB,” *MongoDB*. [Online]. Available: http://www.mongodb.org/. [Accessed: 05-Apr-2014].
[27] “Mongoose,” *Mongoose ODM v3.8.8*. [Online]. Available: http://mongoosejs.com/. [Accessed: 05-Apr-2014].
[28] “Nunjucks,” *Nunjucks*. [Online]. Available: http://jlongster.github.io/nunjucks/. [Accessed: 05-Apr-2014].
[29] “Passport - Simple, unobtrusive authentication for Node.js,” *Passport - Simple, unobtrusive authentication for Node.js*. [Online]. Available: http://passportjs.org. [Accessed: 05-Apr-2014].
[30] “pwd,” *pwd*. [Online]. Available: https://www.npmjs.org/package/pwd. [Accessed: 05-Apr-2014].
[31] “Bootstrap,” *GetBootstrap*. [Online]. Available: http://getbootstrap.com/2.3.2/. [Accessed: 05-Dec-2013].
[32] “Accordion | jQuery UI,” *Accordion | jQuery UI*. [Online]. Available: http://jqueryui.com/accordion/. [Accessed: 05-Apr-2014].
[33] “X-editable :: In place editing with Twitter Bootstrap, jQuery UI or pure jQuery,” *X-editable :: In place editing with Twitter Bootstrap, jQuery UI or pure jQuery*. [Online]. Available: http://vitalets.github.io/x-editable/. [Accessed: 05-Apr-2014].
[34] “D3: Data-Driven Documents,” *D3.js - Data-Driven Documents*. [Online]. Available: http://d3js.org/. [Accessed: 05-Dec-2013].
[35] “jQuery - Write less, do more.,” *jQuery*. [Online]. Available: jquery.com. [Accessed: 05-Dec-2013].
[36] Mike Bostock, “bl.ocks.org - mbostock,” *bl.ocks.org - mbostock*. [Online]. Available: http://bl.ocks.org/mbostock. [Accessed: 05-Apr-2014].

Appendix A:

Previous Report Submissions