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

List of Figures

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:

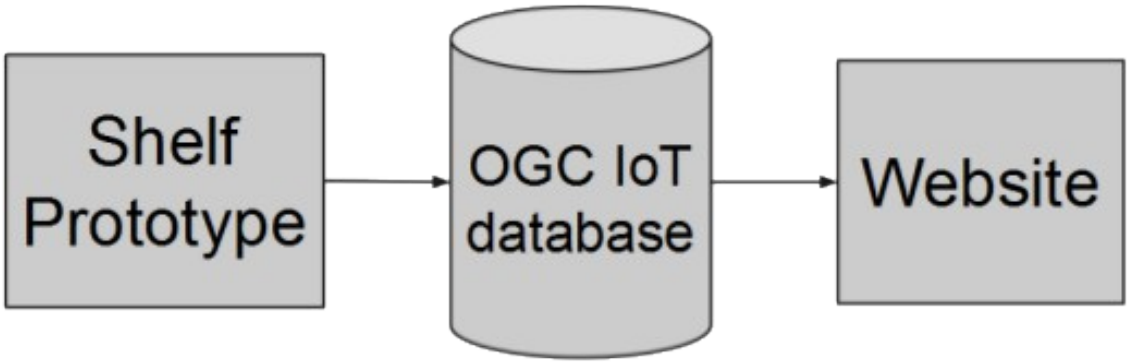

Figure 1: Abstraction of the basic project model

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).

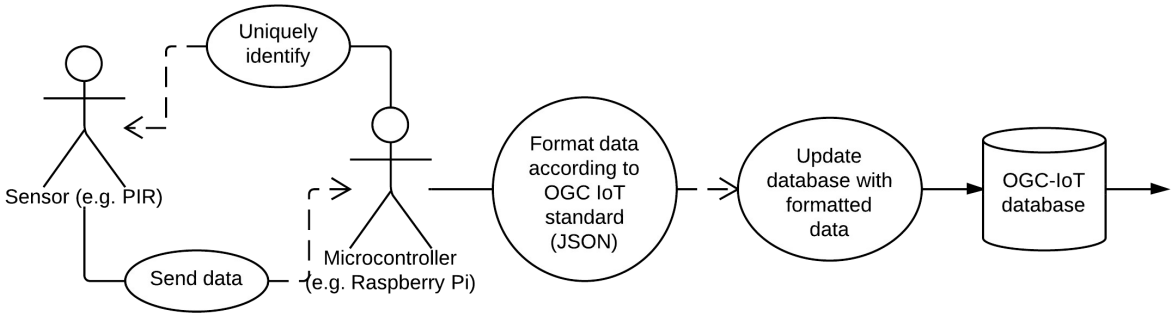

Figure 2: Use Case diagram for Shelf Prototype component of project from Figure 1 above

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:

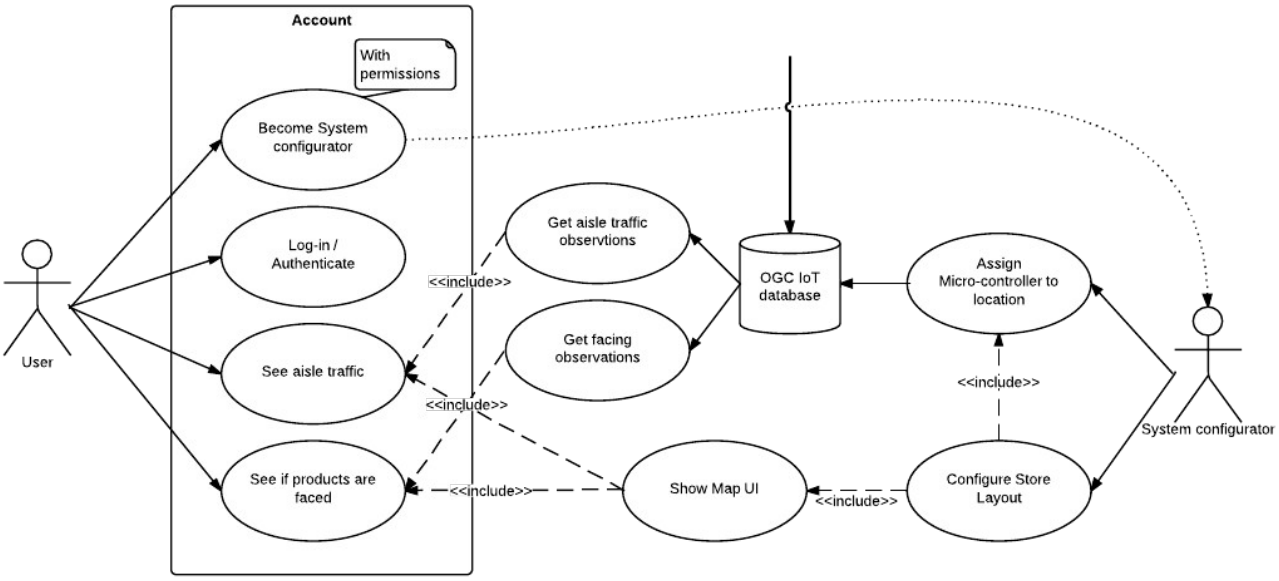

Figure 3: Use Case diagram for Website component of project from Figure 1 above

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.

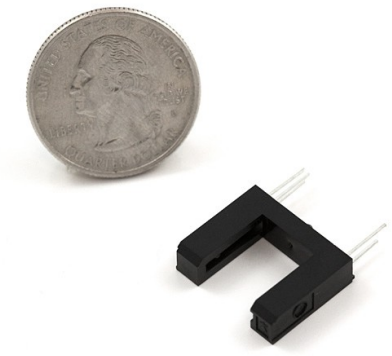

Figure 5: Example of Photo-interrupter from sparkfun.com

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.

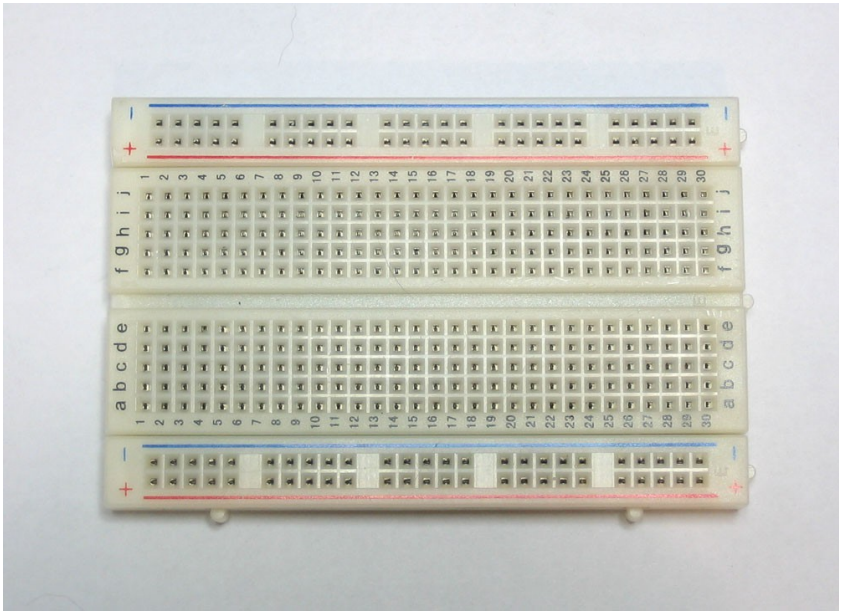

Figure 6: Example of the breadboard we used for testing

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:

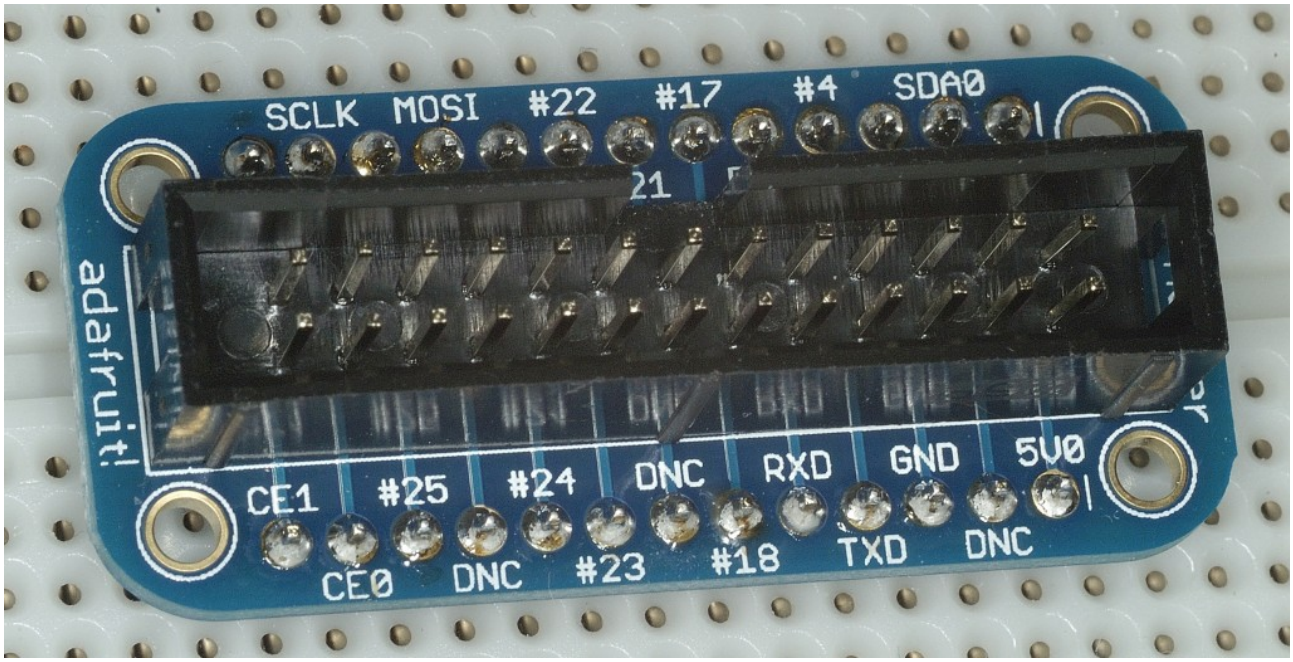

Figure 7: An example of the Pi Cobbler, an accessory for the Raspberry Pi

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.

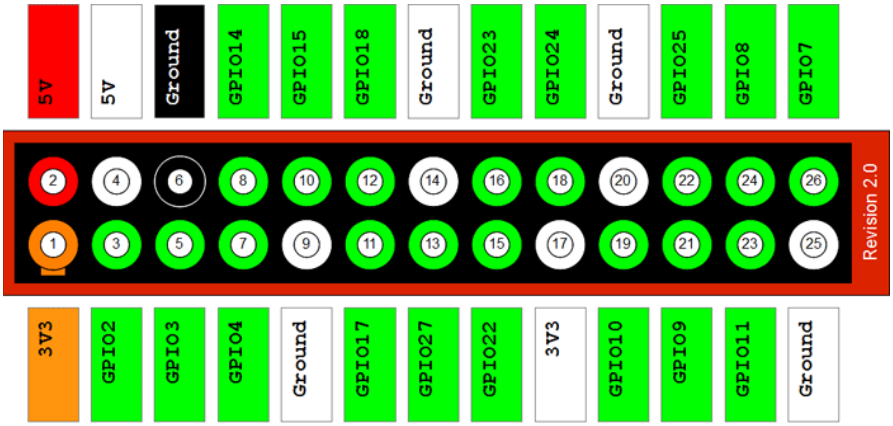

Figure 8: Layout map of the gpio pins on the Raspberry Pi and Pi Cobbler

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:

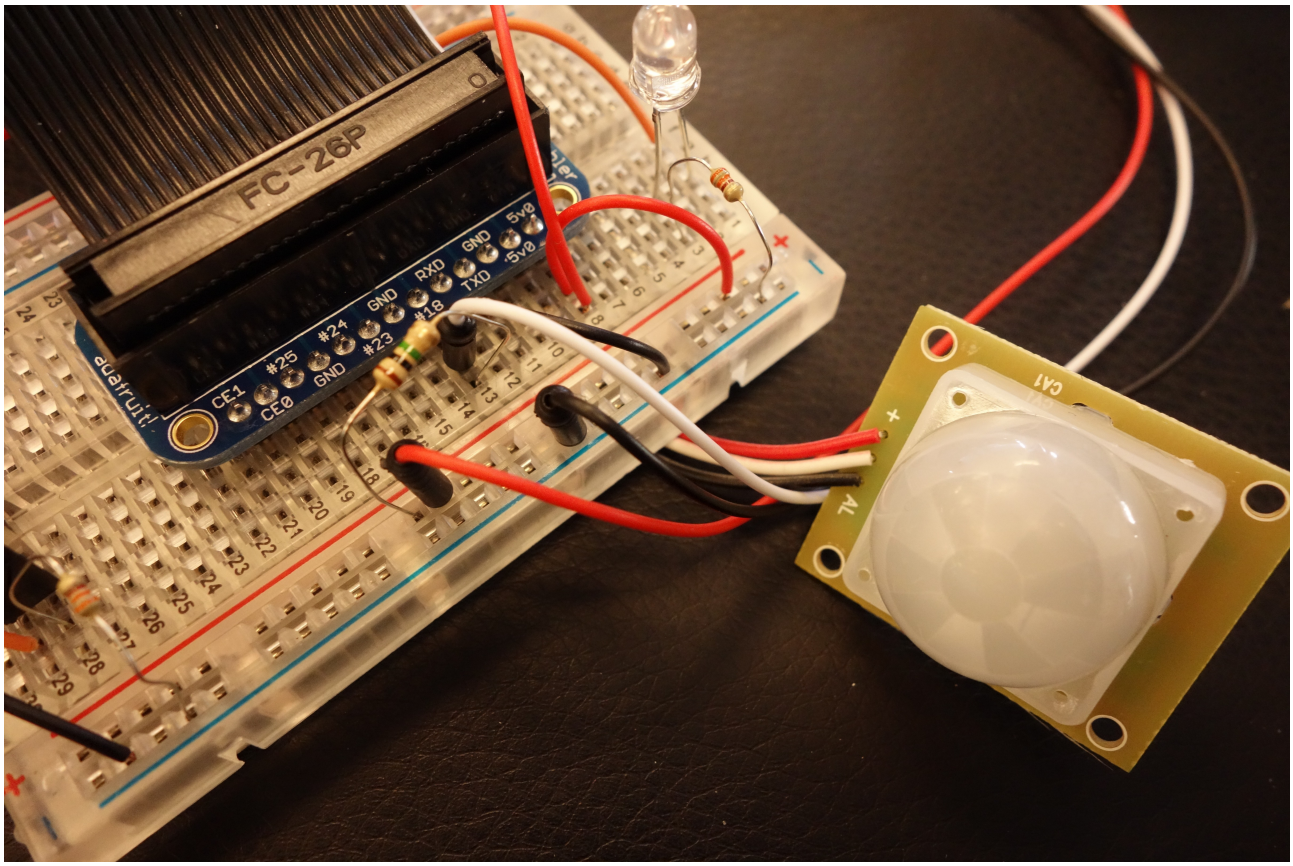

Figure 9: Example of how wiring is set up for PIR Motion Sensor

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

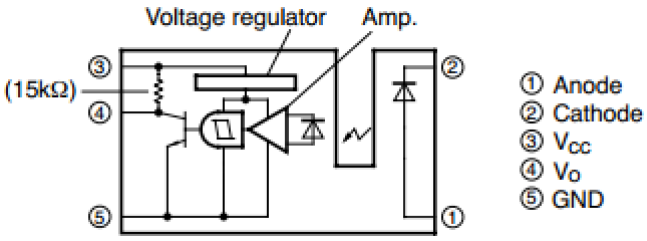

Figure 10: Schematic Diagram of Photo-interrupter

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:

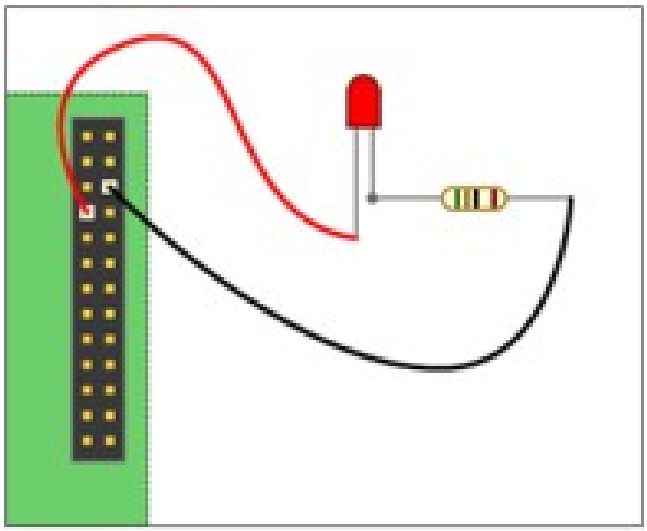

Figure 11: Schematic of how we wired the LED with a resistor

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.

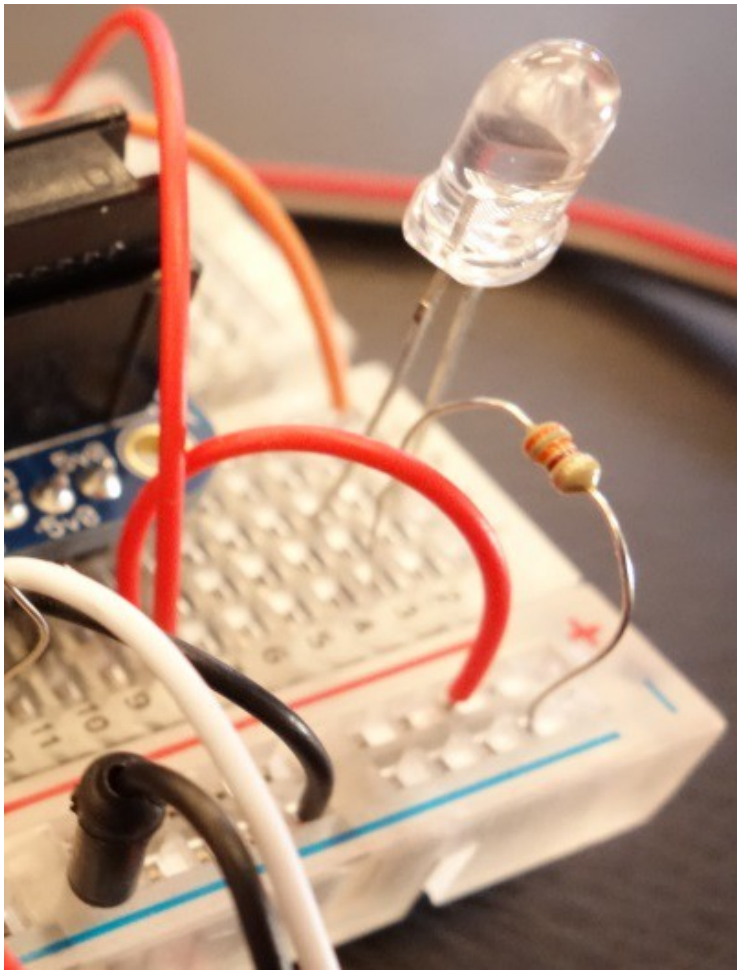

Figure 12: Photo of how we set up the LED on the breadboard

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:

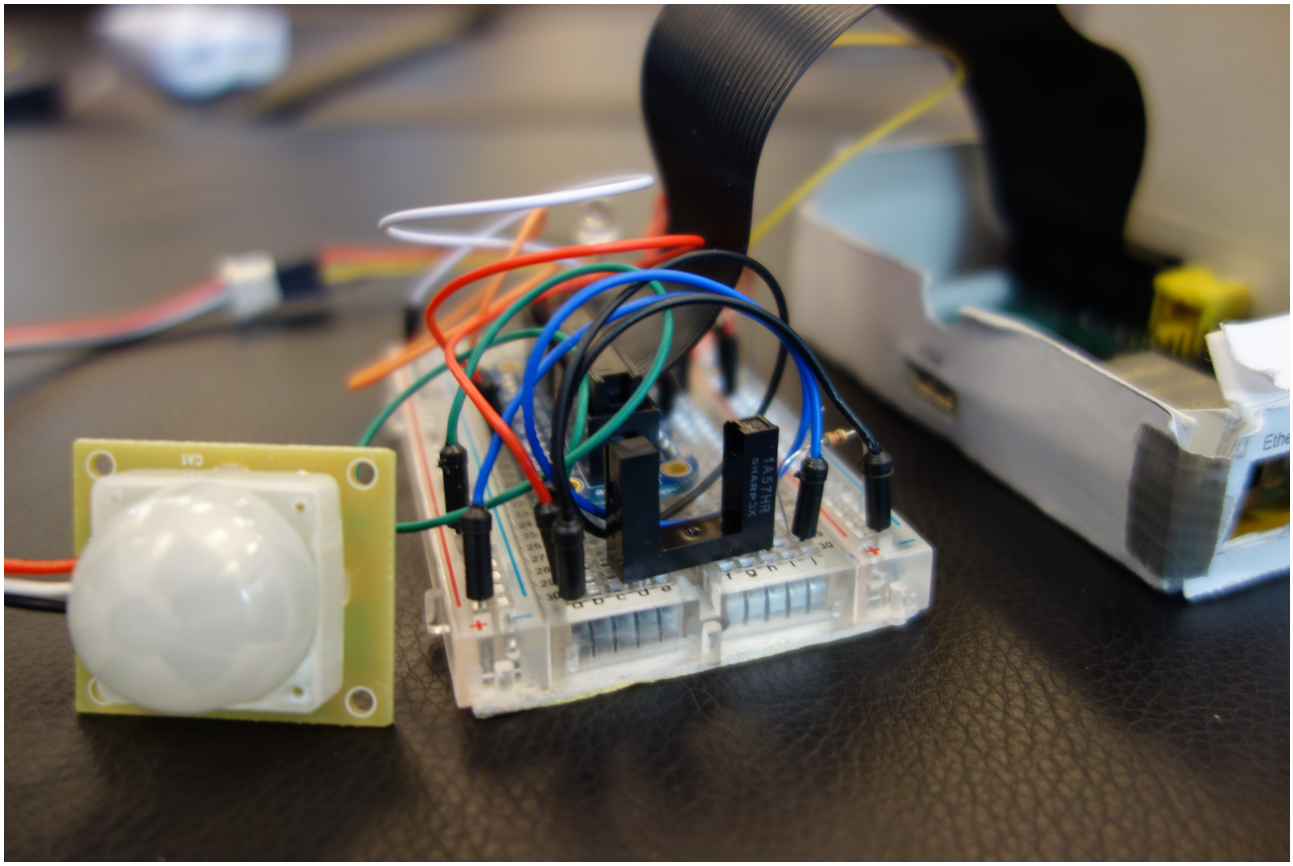

Figure 13: Example set up of all the sensors and components together

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.

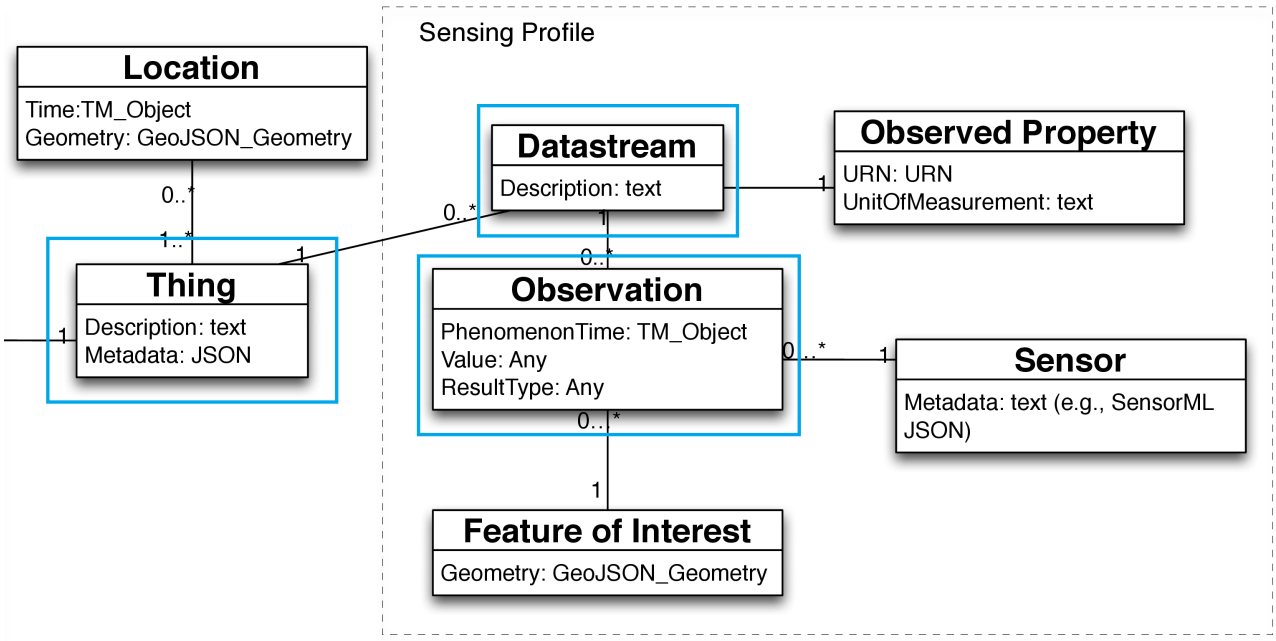

Figure 14: Data model defined by the OGC

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.

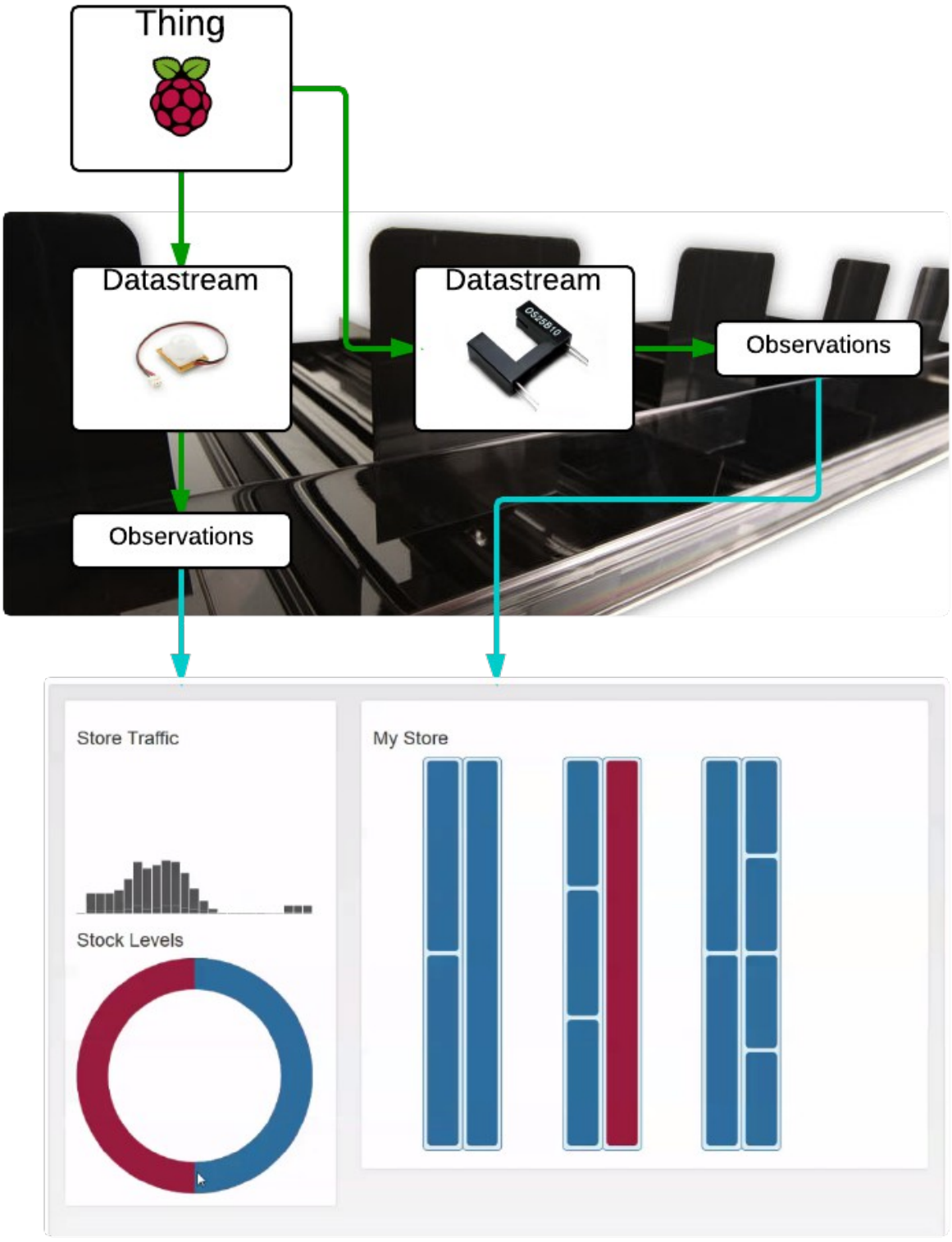

Figure 15: Adopted data model we used for our design

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:

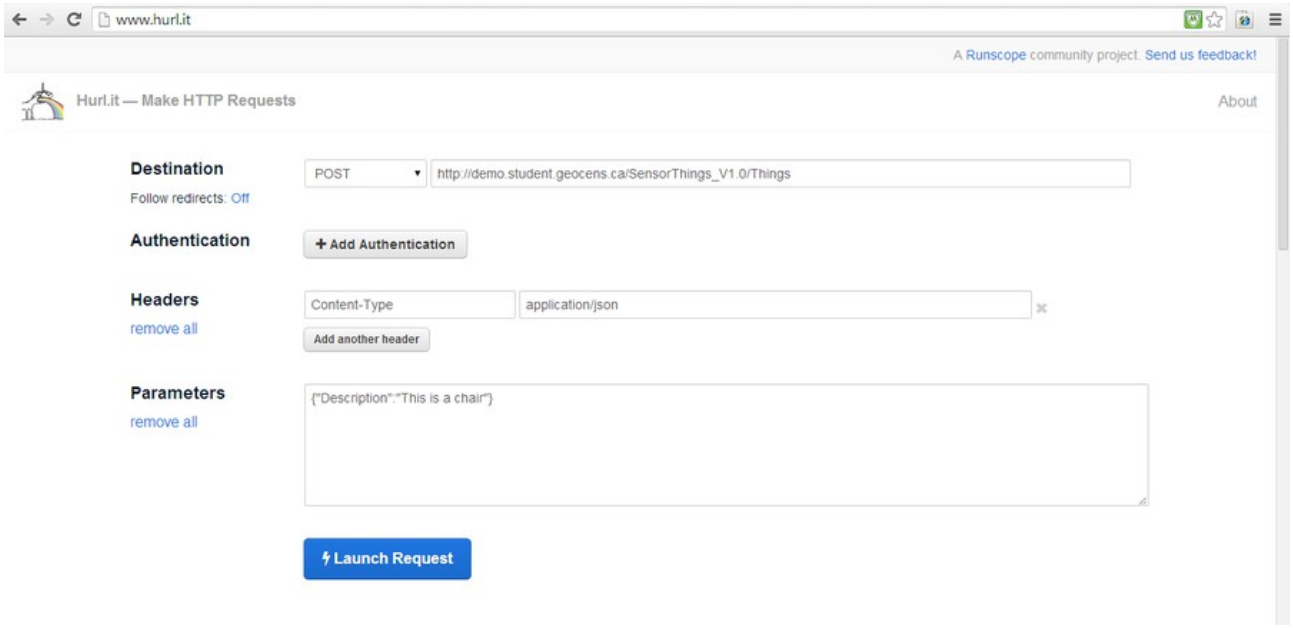

Figure 16: Screenshot of what the hurl request should appear like

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

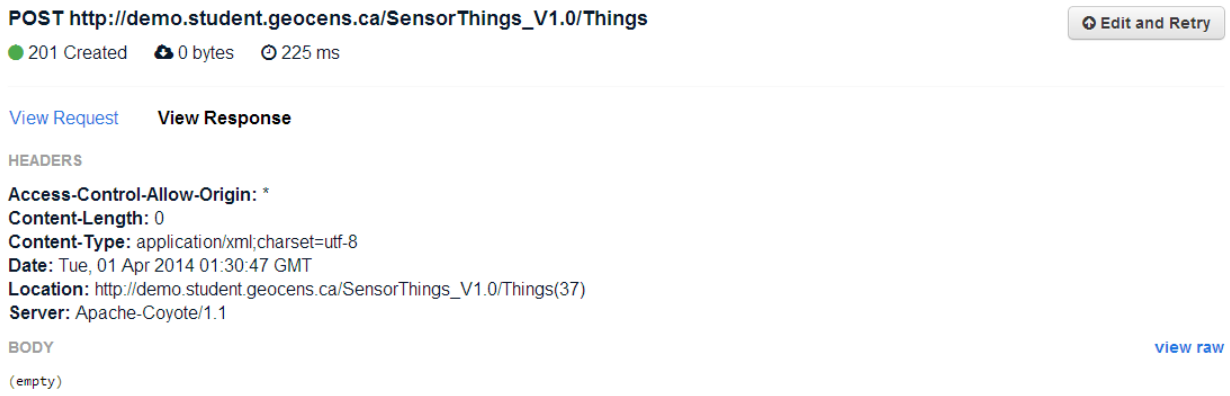

Figure 17: Result from sending a POST request to the SensorThings API

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).

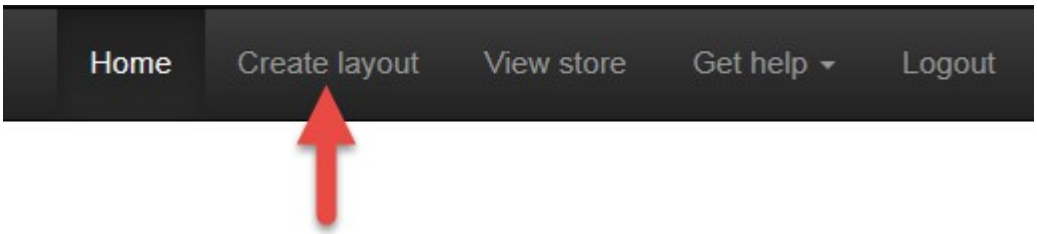

Figure 18: Screenshot of the navigation bar

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

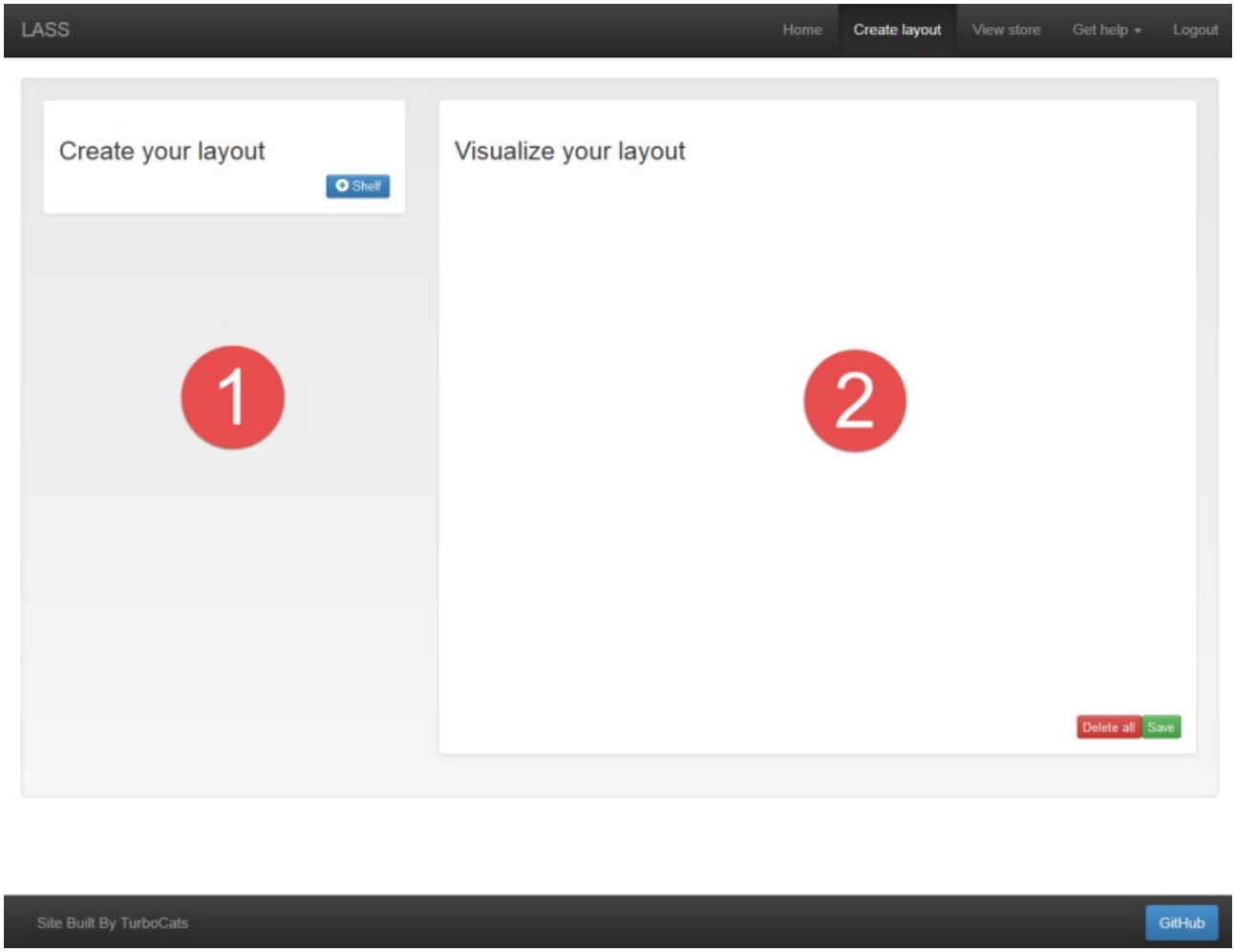

Figure 19: Initial page a user will see if they have an empty store layout (have yet to create one)

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.

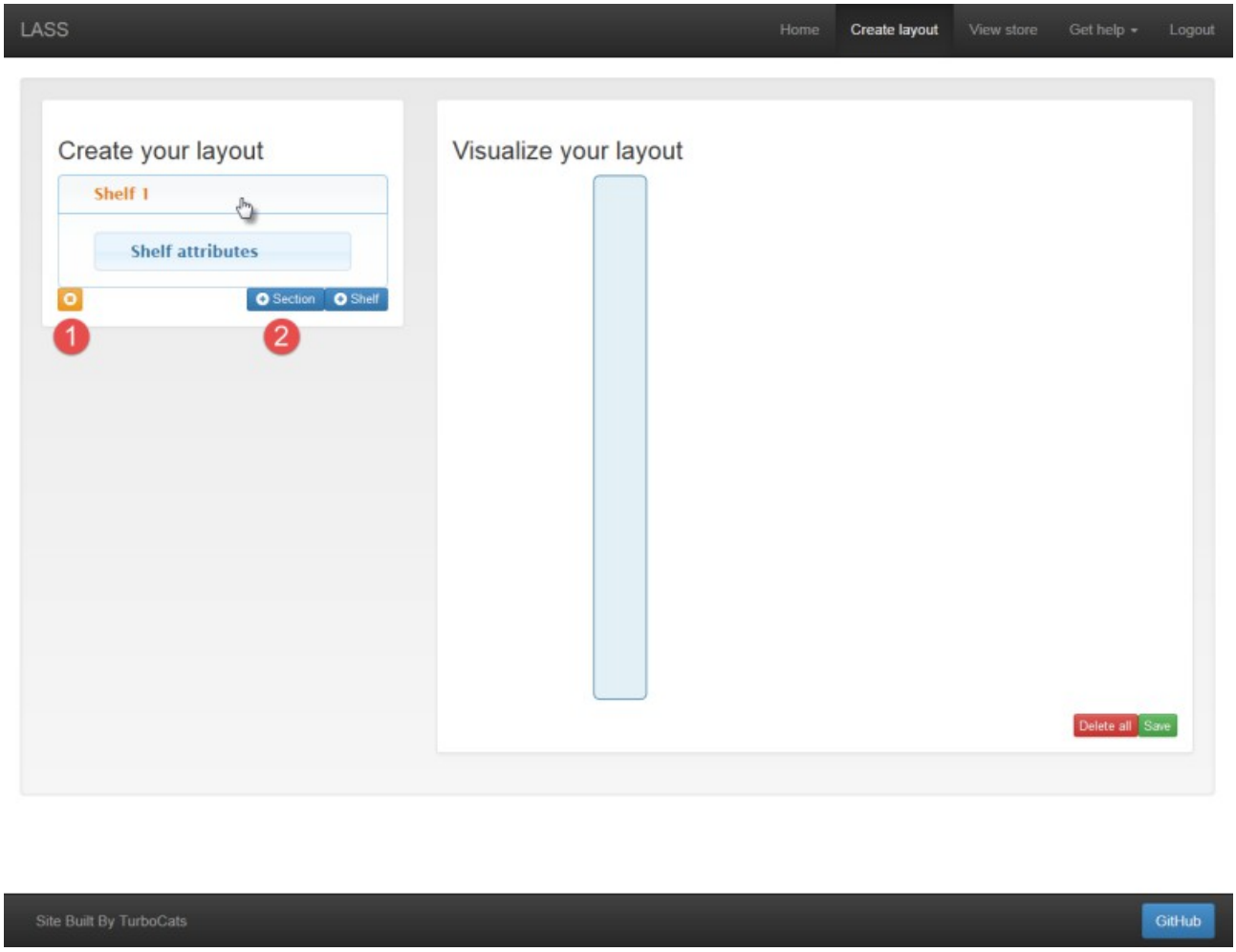

Figure 20: Depiction of when a shelf is added to the store layout

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.

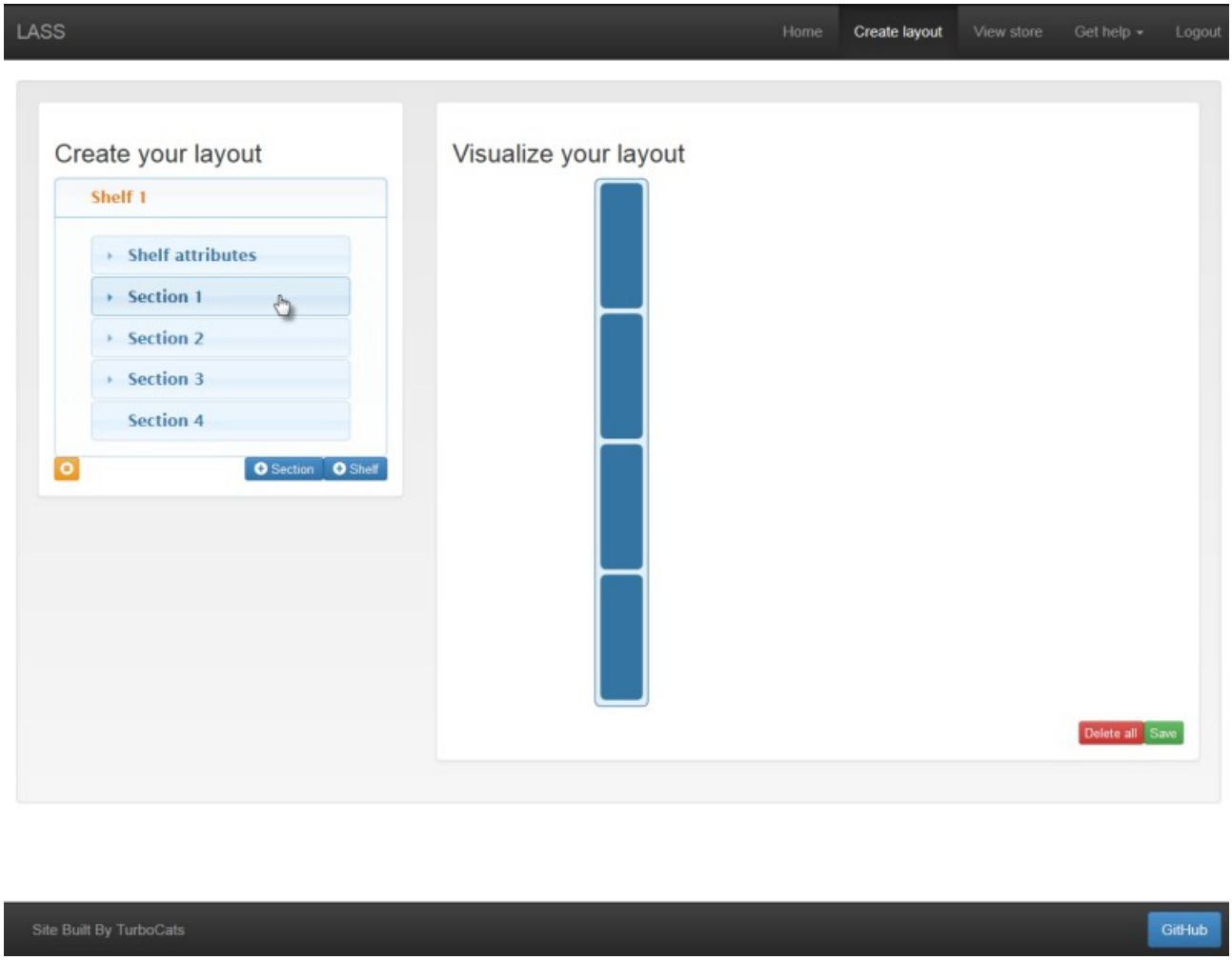

Figure 21: Depiction of what happens as you add sections to your shelves

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.

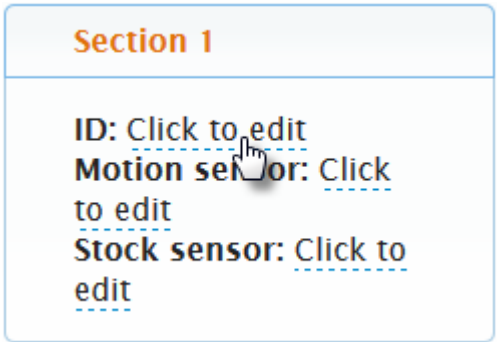

Figure 22: A single section with editable attributes

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.

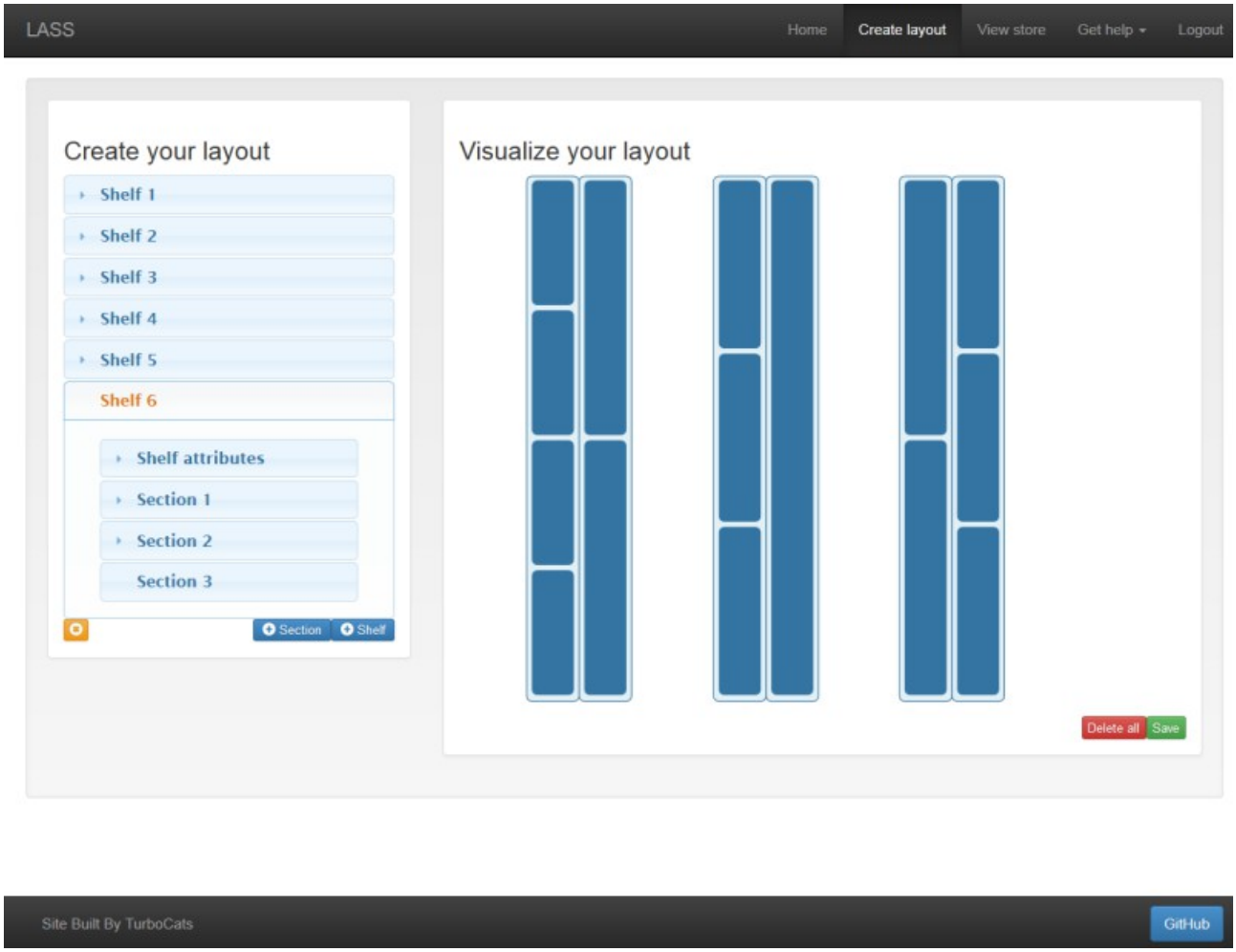

Figure 23: Complete Store Layout

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:

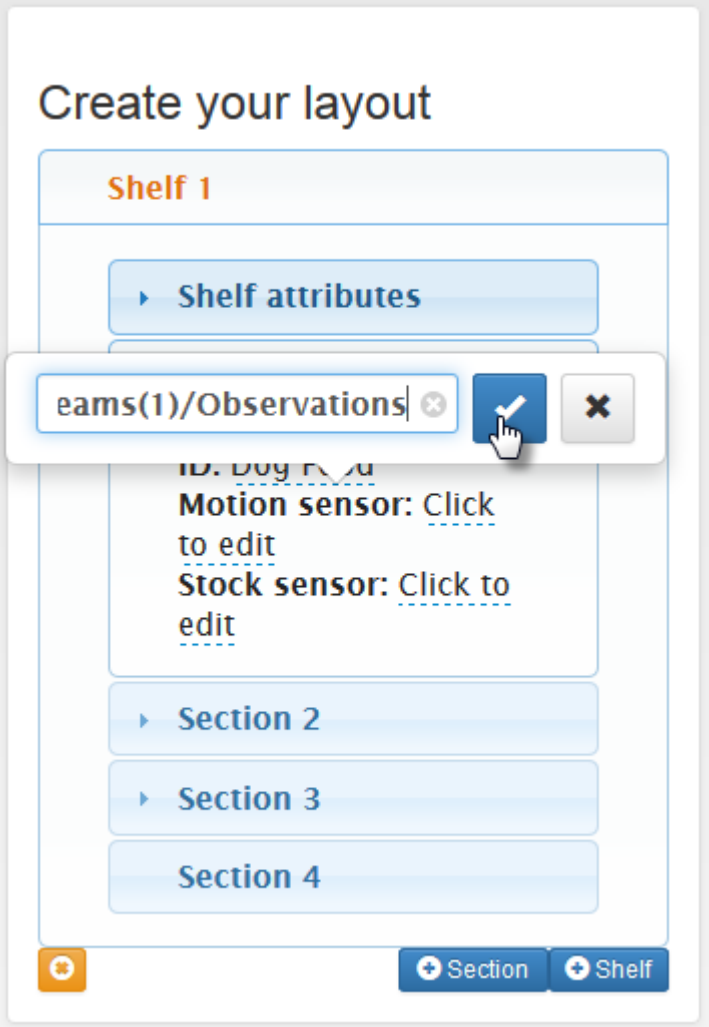

Figure 24: Linking Section to Data Model

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:

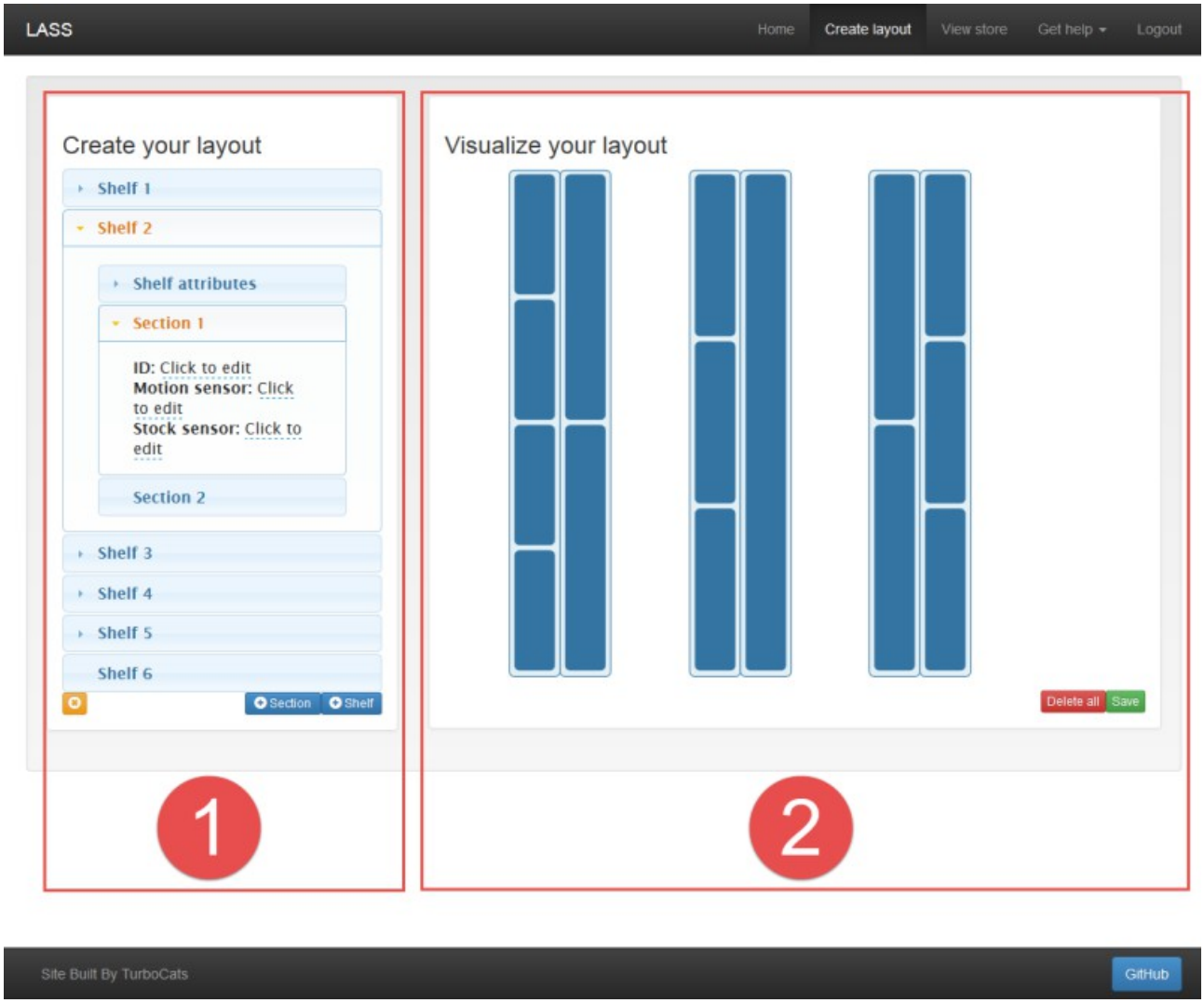

Figure 25: Screenshot of the layout creator in action

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:

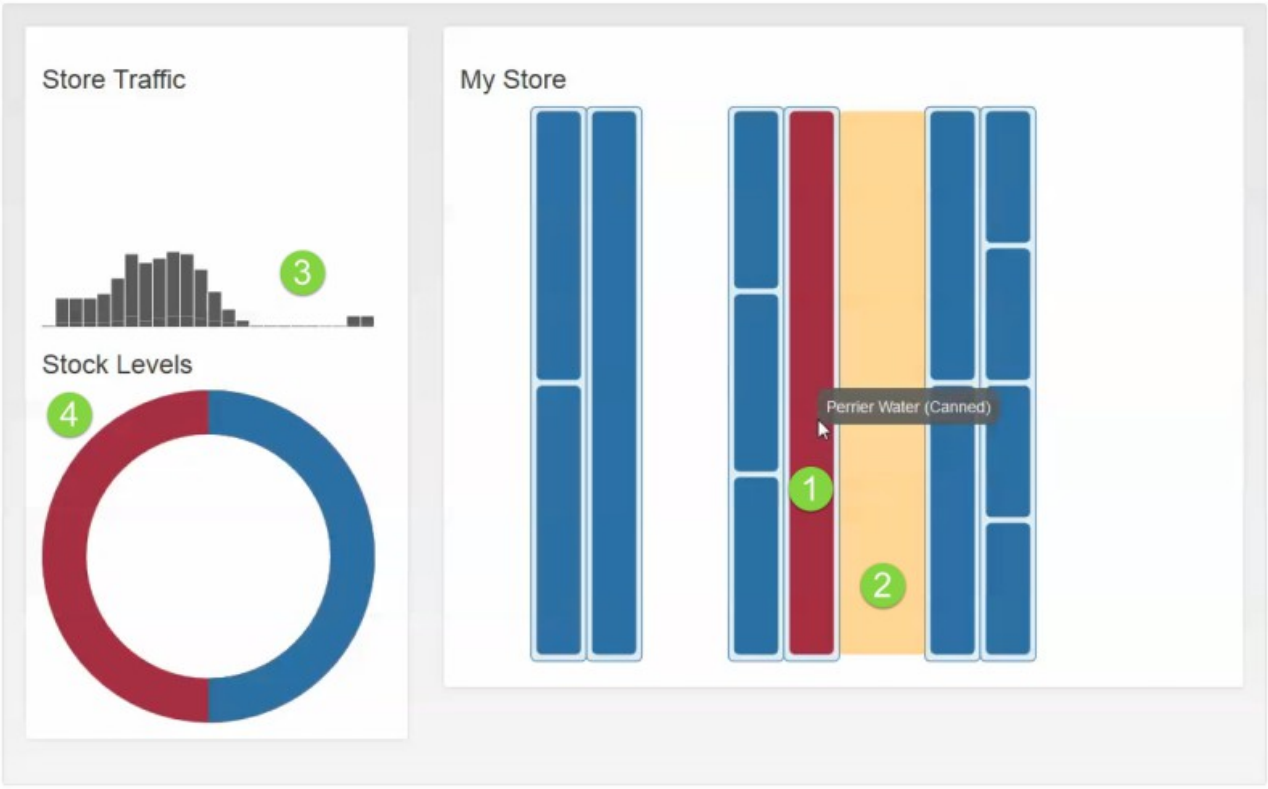

Figure 26: Screenshot of system working with active observations being displayed

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:

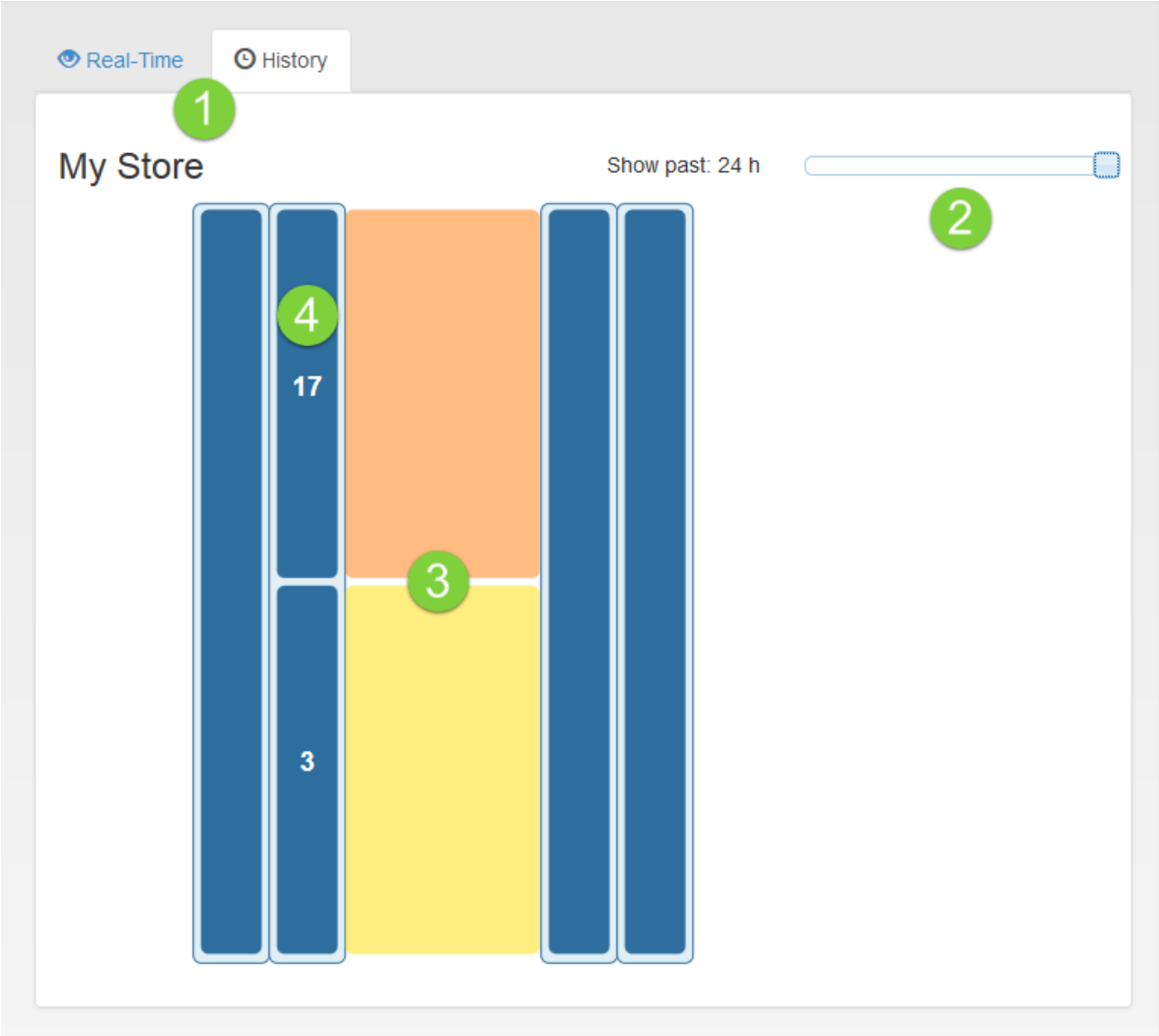

Figure 27: Example of viewing historical data for your store

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

UNIVERSITY OF CALGARY

ENGO 500: GIS & Land Tenure 2

Project Proposal

By: Jeremy Steward
Kathleen Ang
Ben Trodd
Harshini Nanduri
Alexandra Cummins

Supervisor: Dr. Steve Liang

DEPARTMENT OF GEOMATICS ENGINEERING

SCHULICH SCHOOL OF ENGINEERING

09/27/2013

Table of Contents

Index of Figures

Index of Tables

Introduction

The revolution of the Internet changed the future of communication forever. As time went by, new technologies were developed which allowed us to move from the First Generation of the Internet to the Fourth Generation. In the 1970s the First Generation of Internet began through APRAnet (Advanced Research Projects Agency Network). In the 1990s the Second Generation of Internet was accompanied by AOL. In the 2000s, the third generation began and we are still in it today. This generation is dominated by social media, such as Facebook, Twitter, MySpace, Flickr, etc. Society is now in the process of transitioning from the Third to Fourth Generation. The Fourth generation of Internet is represented by the Internet of Things (IoT) [1].

IoT is a new concept that involves the networking of all physical devices to function more cohesively as one unit to assist in everyday life. It connects everything to the internet “virtually”. IoT will have a huge impact on businesses all over the world as well as the average household. Some advantages of IoT include: increase in household security, increase in energy conservation and also an increase in business sales [1]. However, the potential for such a complex network of connections also leads to many different implementations. Because of this, it is desirable to have an accepted standard for application development.

The Open Geospatial Consortium (OGC) is an international consortium with more than 445 companies, research organizations, government agencies, and universities that participate in a consensus process to develop geospatial standards that are available publicly. The OGC Standards allow technology developers to make geospatial information and services useful with any application that needs to be geospatially enabled [2]. According to Dr. Steve Liang, Sensor Web for IoT convener and director of GeoSensorWeb Laboratory at the University of Calgary, "The standards coming out of this OGC process will make it possible for developers to ensure that sensors, the observations they produce and the systems they inform will be easy to reach and control with Web services, without compromising security and data integrity” [2].

This project involves using sensors in order to develop an Internet of Things application using the Open Geospatial Consortium (OGC) standard. The project is designed within the scope of ENGO 500, Geomatics Engineering Project. Our aim for this course is to work on a design project as a group in order to increase and apply our knowledge to a real-world problem, which will prepare us to work as professional engineers in the future.

There are a few goals that we wish to fulfill by the end of the project. The first goal is to design and plan a project, and complete it successfully by working as a team. Our second goal is to produce an open-source web-service application that uses IoT to solve problems that are relevant to the population in Calgary. The third goal is to learn more about the OGC standards for IoT, and to develop a prototype that uses the specifications that were outlined by the OGC committee. Our final goal is to improve our project management skills by using Github. Github provides a way to keep track of tasks assigned to each individual and all the documentation/reports required to successfully complete this project.

Background Information

Internet of Things

One of the primary aims of this project is to develop an application of the IoT. The term ‘Internet of

Things' (IoT) refers to the network formed between tangible objects and their virtual counterparts, and is the result of linking these two representations via an internet like structure. This is usually done by equipping or embedding objects with sensors, providing them with a unique identifier and the ability to communicate without requiring human interaction [3].

Objects used in an IoT may be anything from household appliances to living creatures. In each of these cases, data from the sensor assigned to an object is transferred over a network and may be accessed in real time, in order to gain insight on temperature, noise levels or other measurable conditions of the object being observed.

Open Geospatial Consortium

The Open Geospatial Consortium (OGC) is a non-profit international organization, and exists as a medium though which more than 400 organizations worldwide may collaborate to encourage development and implementation of open standards for geospatial services, data sharing and geographic information system (GIS) data processing [4]. Founded in 1994, the main focus of the OGC is the creation of a set of technical documents known as OGC Standards. These documents detail encodings, which ensure complex spatial information remains available and relevant to various applications. OGC Standards are developed by a unique consensus process, which ultimately allow geoprocessing technologies to interoperate, thus guaranteeing compatibility between two or more separate products [5].

OGC RESTful API

There are two different types of web services, the first being Simple Object Access Protocol (SOAP) based services, and the second being Representational State Transfer (REST) based services. The primary difference between the two types of services is that SOAP-based services transfer data in the form of XML documents through a specified schema, while REST-based systems transfer data by dynamically generating web pages (e.g. through some Common Gateway Interface (CGI) based application) [6]. An Application Programming Interface (API) exists to aid developers in writing applications by providing them with an abstraction layer that specifies how software components such as data or processes should request services from the underlying hardware or software layers [7]. In the context of web services such as SOAP or a RESTful API, as we intend to use in this project, it usually consists of a set of classes that can format and make requests from the web service, and thus simplifies the use of the service [7]. Effectively this provides us a library or a suite of functions and objects with which to interact with the web service.

This particular project will likely use multiple devices or sensors. While the developers of these sensors or devices may have provided a unique API for each sensor, this will likely bring difficulties when integrating more than one sensor or data set. Therefore, there is a need for a single, standards-compliant API, that can access and interact with any sensor we may choose to use. Linking everything through a common API will facilitate the project in a number of ways: it will centralize and ease the connection process to and from the sensors (this includes reducing connection and execution time of the application, as well as necessary power), it will simplify the code (thereby reducing the number of various languages and models required for developers to learn in order to use any specific sensor), and finally it will ease the task of maintaining security for one web service rather than several. Finally, it will be easier for us as the developers to implement one API rather than several over the duration of the project.

To implement our project, we are planning to use the RESTful Web API design model, or RESTful

web service. This is an architectural-style web API, implemented using HTTP and REST principles. A RESTful API service requires several necessary aspects, including the media type of the data supported by the web API, and a set of operations (such as GET, PUT, or DELETE) using HTTP methods as mentioned above [8]. In addition, these aspects should include some base Uniform Resource Identifier (URI) string for the Web API, as we want to be able to identify our data as we are accessing it [9].

Objectives

The overall goal of this project is to use the Internet of Things (IoT) to tangibly assist the citizens of the city of Calgary in a practical and user-friendly way. In so doing, the end product should utilize the OGC RESTful API and should also be open source. In order to accomplish this goal, several smaller objectives and sub-objectives have been set out. The intention of these objectives is to set out clear, achievable tasks which will lead to a useful application and also fulfill the requirements of this course. However, because these are being outlined at a very early stage in the project, it is anticipated that some adjustments will occur as the project progresses. The following list describes each of the main objectives and their subsequent sub-objectives.

1. Determine a location-based application of the IoT which is relevant to the city of Calgary. This requires defining a problem relevant to Calgarians (or a particular sub-group of citizens, such as civil servants, university students, environmentalists, etc.), specifically identifying a component which can be measured and the importance of its spatial properties. The scope of the application should be outlined, including what area it will cover and what value it will bring. In order to continue with the project, the application should be selected by October 4, 2013.

2. Design a sensor setup which can measure data suitable for the application determined in objective 1.

    a) Determine the technical requirements necessary for data measurement, such as the sensor(s) required, and other periphery elements needed. These requirements should be written into a technical specification report with an accompanying sketch of the sensor setup.

    b) Create a working prototype of a sensor which could be deployed in the city. The prototype’s technical specifications should follow the technical requirements detailed in sub-objective 2a.

    c) Test the prototype in a lab setting, in order to understand functionality and limitations. Simulated environments should be used so that adjustments can be made before testing in a field setting. All test data should be saved and stored in a secure format, and should be accompanied with detailed experimental notes (such as dates of testing, differences in conditions, etc.)

3. Develop a library which links the sensor setup and the OGC RESTful API.

    a) Learn necessary language(s) required for writing a library which will work with each sensor in the chosen sensor setup. Proficiency is needed to a point where it will be possible to write a working library; as such, learning should have a concentrated focus.

    b) Collect set of test data with multiple sensors in a working/usable configuration in the

city. The configuration may be sparser than actual deployment, but should be a reasonable (e.g. at least 5 sensors) network. All test data should be recorded and stored in a secure format. Experimental notes should be kept, including details such as date(s) of experimentation and times, pictures of location, weather conditions, etc.

4. Create a user interface (UI) such as a web site or mobile application which makes use of the OGC RESTful API as provided by Dr. Liang's research group to display the sensor data.

    a) Design a user-friendly graphical interface which is tested and found intuitive by at least 80% of test subjects. Testing will be based on hands-on experience and will be evaluated by means of a standardized survey. Feedback collected will be used to improve the final version of the user interface.

    b) Represent collected data (in real time) using the created user interface and create a video of its usage. The video should show both how to use the developed UI, as well as prove its functionality.

5. Fulfill all deliverable obligations as explained in the ENGO 500 course outline punctually and professionally. These deliverables will also serve as one means for tangible documentation and as a measure of progress.

    a) Write a project proposal which outlines background information, project objectives, methods and expected outcomes by September 27, 2013. This proposal should include some initial research into the topic, as well as planned objectives and methods for meeting those objectives. A Gantt chart showing the planned breakdown of tasks should also be included.

    b) Complete a literature review which thoroughly explores the history of the topic, current state-of-the-art and applications. This should be completed by November 18, 2013.

    c) Present a report of technical deliverables in two forms: written and oral. The written report component should be accomplished by December 6, 2013 and the oral component should be presented at an appointed time between November 22 and December 6, 2013. This report should cover the technical deliverables which are to be accomplished in the previous four objectives.

    d) Give a progress report in two formats: written and oral. The written report should be completed by February 14, 2014 and the oral report should be given at an assigned time between January 31 and February 14, 2014. The progress report will communicate work done towards the final goal.

    e) Create a final project report in two formats: written and oral. The first draft of the written final report will be completed by April 4, 2014. An oral report, given as a defense of the written report, will take place at an appointed time between April 7-11, 2014. Based on feedback from the oral defense, a revised copy of the final report will be finished by April 28, 2014. The final report represents the culmination of all the project work throughout the school year.

    f) Present at the Capstone design fair. The date has yet to be announced, but the expected date should fall somewhere around the end of March or April. This will be used to showcase the project's final products.

g) Upload and copy documentation as well as all necessary work onto the group's Github repository [10]. This repository will serve as a means of management for the group, and is also fulfills the goal of being open source. This will serve to assist any future work that may be performed using this project as a base.

Methods

Sensors

Computer Board

All sensors will be interfaced using a single board computer. These computers are well suited for do-it-yourself (DIY), educational, and prototyping projects due to their low cost and the wide variety of available sensors. The boards considered for the project are the Raspberry Pi and the Arduino Uno. Although they are similar in size and cost, their intended purposes differ slightly. The Raspberry Pi is a fully operational computer running a Linux operating system [11]. Having a full OS gives the Raspberry Pi the luxury of communicating with higher level devices such as a USB WiFi dongle or sensors with digital output. Conversely, the Arduino Uno is a micro-controller designed for embedded applications, meaning there is no operating system [12]. The Arduino is well suited to communicating with low level sensors which give either analog or digital signals. Table 1 below highlights their differences:

Table 1: Specifications for Arduino Uno, and Models A and B of the Raspberry Pi. Information taken from [11,12,13]

Name	Arduino Uno	Raspberry Pi Model A	Raspberry Pi Model B
Cost (USD)	$29.95	Discontinued	$35.00
Size	2.5”x2.10”	3.37”x2.125”	3.37”x2.125”
Processor	ATMega 328 @ 16MHz	ARM11 @ 750MHz	ARM11 @ 750MHz
SRAM	2 KB	256 MB (Shared w GPU)	512 MB (Shared w GPU)
Flash Memory	32 KB	SD CARD	SD CARD
Ethernet	None	None	Ethernet
WiFi	Adapter available	3rd party adapter	3rd party adapter
Digital GP I/O	14	8	8
Analog Input	6	None	None
I²C	2	1	1
SPI	1	1	1
USB	None	1	2
Video Out	None	HDMI, Composite RCA	HDMI, Composite RCA
Audio Out	None	HDMI, 3.5 mm jack Analog	HDMI, 3.5 mm jack Analog
Input Voltage	7-12v	5v	5v
Power Requirements	42 mA	300 mA	700 mA

Below we have listed some of the advantages of each of the boards.

Raspberry Pi:

- Internet connectivity: The Raspberry Pi includes a full GNU/Linux OS and therefore includes advanced networking capabilities. It provides an Ethernet port as well as plug and

play support for some USB Wifi & 3G dongles.

- Performance: While the Raspberry Pi does have the overhead of running an OS, it boasts a 750 MHz processor which can be overclocked up to 1GHz with either 256 MB or 512 MB (shared with GPU) of RAM available. The flash memory is also larger (depending on the size of SD card used).

- Cost: While the board itself is comparable to the Arduino, being able to use a standard USB WiFi dongle is a great advantage over the Arduino WiFi shield (~$10 vs $70).

- Output: If the sensor needs to be controlled in-situ, common video and audio output formats will help accomplish this.

- Programming Flexibility: With a full GNU/Linux OS, any language that compiles on ARMv6 architecture can be used. This opens up the possibility of using many languages or a mix of languages.

Arduino Uno:

- Power requirements: If the application requires batteries as the power source, the Arduino uses significantly less power than the Raspberry Pi.

- Sensor compatibility: The Arduino offers more ports to interface with sensors outputting both digital and analog signals. Comparatively the Raspberry Pi cannot communicate with analog output sensors.

- Large development community: The Arduino development community is more mature than the Raspberry Pi's with a large number of tutorial and example projects [13].

While other computer boards may be considered depending on the application we decide upon, many are comparable to these two boards, and as such only these boards will be explored for the remainder of this report.

Other Sensors Considered

For the location of the device, the sensor set-up will likely be one of two situations:

- If the hardware was designed to be used in an application where the sensor is moving, a GPS receiver will likely be included. Adding a GPS receiver will increase project cost by means of the receiver as well as its power supply.

- If the sensor is designed to gather spatial data (i.e. image based) from a discrete location, it could be possible to avoid including a GPS receiver if the location of the sensors does not have to be known to a high precision.

The other sensors to be included on the device depend on the application chosen. There are a wide variety of sensors available, many of which are inexpensive. Examples of other sensors that could be include are: image, video, sound, temperature, pressure, humidity, altitude, distance, magnetometer, accelerometer, force sensitive resistor, tilt, infrared, luminosity, motion, radiation, liquid flow, stretch and more [14].

Interfacing with the Database

The sensor assembly will have network access of some kind, with which it will send sensor data to a database designed specifically for IoT applications. This server is maintained by the Sensor Web Interface for IoT Standard working group [15]. The sensor assembly will require a library or set of libraries to interface with the database's OGC RESTful API, which will be developed to adhere to the upcoming OGC Standard Sensor Web interface for IOT (SWIOT) [16] standard. This project will likely be one of the first to use the SWIOT standard, and it is hoped that the project helps the standard get recognized and used by other developers.

Programming Language(s)

The Raspberry Pi is configured to support Python by default, but it is easy to use other languages such as C/C++ or anything else that will compile on ARMv6 architecture. Advantages of using Python are that as high level programming language it allows for easy development and smaller source size. Due to its interpreted nature, it is slower and uses more resources than C++. Advantages of using C/C++ are that as an intermediate programming language it is faster and more flexible than Python. Development of C++ programs is more complex, which takes more time and code is typically 5-10x larger in C++ [17].

The Arduino offers its own integrated development environment (IDE) where the device can be programmed using the Arduino Programming Language, based on C/C++. It also has many libraries available by default for connecting to standard sensors [18].

Organization / Management

The project will be managed through Github which will serve as a means to plan & document the project during its evolution. Github has many tools to aid projects along their development and is a good choice for the project due to its open source nature. The Github tools that will be used throughout the project's lifetime are:

- **Milestones:** For any deliverables required or goals set during the project, a milestone will be created. A milestone can be given a due date, and will include one or more tasks that need to be accomplished.

- **Issues:** Any time there is a task that needs doing, an Issue will be created for it. The issue's description will outline background information, objectives, and can be associated with a milestone. Issues can be assigned to users or users can volunteer by assigning an issue to themselves. Once an issue is assigned to a user, other user's know not to work on it without addressing the issue’s owner beforehand. Issues can be labeled and sorted in various ways to help manage the backlog, and will serve as the main organizational system for our day to day work.

- **Wiki:** The Github wiki will serve as the main repository for all the knowledge amassed throughout the project. It is readable by anyone but only project collaborators (our group) can add to it for the time being. By compiling all data in one place all team members can stay current with the project's evolution without needing to consult multiple sources.

- **Pulling/Forking:** Github uses the content tracking system git to track changes to files (usually code), often by means of *forking* and *pulling*. A user can fork (or copy) a repository

(collection of files) and then subsequently create a local copy on their machine. From here they can make any edits they desire, and if they want to commit the changes, they create a pull request. This pull request is reviewed by other teammates, specifically the group leader, and if accepted then these changes are merged into the master branch of the main repository. The group will likely only work on the master branch for the duration of the project, but if necessary other branches might be developed and explored.

By utilizing these tools the group hopes to manage the project in an open, detailed, and organized manner in which all group members are accountable and progress is maintained.

Timeline

See the Gantt chart in Figure 1 on the next page to see the task breakdown and expected timeline for milestones and objectives throughout the project. Note that the project runs from week 0 (week of September 13th, 2013) until week 32 (week of April 25th, 2013).

Figure 1: Gantt Chart of Project Milestones and Tasks

Conclusion

The desired outcomes of this project are to:

1. Design and plan a project, and to carry the project through to completion working in a team environment [19].

2. Produce an open source web-service application (web site or mobile application) that uses the Internet of Things (IoT) to solve a problem relevant to us locally (i.e. in Calgary). By open sourcing this application, we hope to provide some meaningful benefit to either Calgarians or any other entity that may wish to take over the project upon its completion at the end of the year.

3. Learn more about the Open Geospatial Consortium (OGC) standards for IoT, and develop a prototype application that uses the upcoming standard specification as outlined by the OGC committee. This application, which should match our second outcome, should ideally serve as an example for those who wish to use or implement the specification in the future, once it has been fully published.

4. Learn about project management skills, in particular managing projects using Github.

Ideally, by meeting these outcomes, we will fulfill a greater outcome, that is, to have worked on an independent project as a group to further our own knowledge and prepare ourselves to work as professional engineers. Likewise, while many smaller outcomes may be fulfilled, such as learning more about programming, sensor integration through micro-controller boards, and others, these are merely subsets of the main outcomes listed above.

In order to realise these outcomes, we will work over the course of 32 weeks (both fall and winter semesters) on developing a web site or mobile application that integrates sensor data using the OGC IoT standard. To accomplish this, we will first determine a relevant problem to us locally, and develop a sensor network of some fashion to use as a means to solve this problem. We will then use data from the sensors and develop an API layer to communicate with the OGC standards-compliant database that Dr. Liang has provided. Finally, we will use the database as the basis to develop our web site or mobile application, as listed in outcome 2, which should either manipulate or provide information from the sensors that targets our problem.

In the process of completing all of this, we will seek to improve our project management skills and documentation abilities by keeping track of the project through Github and completing the required reports for the course. By working under constrained deadlines, and by providing documentation as we develop the project, we will ultimately be working closer towards becoming professional engineers ourselves.

References

[1] Suny Cortland, "The Internet of Things," [Online]. Available: https://sites.google.com/a/cortland.edu/the-internet-of-things/home. [Accessed 22 September 2013].

[2] "The OGC Forms "Sensor Web for IoT" Standards Working Group," [Online]. Available: http://www.opengeospatial.org/node/1650. [Accessed 22 September 2013].

[3] I. Wigmore, "Internet Of Things (IoT)," July 2013. [Online]. Available: http://whatis.techtarget.com/definition/Internet-of-Things. [Accessed 23 September 2013].

[4] "About OGC," Open Geospatial Consortium, 1994. [Online]. Available: http://www.opengeospatial.org/ogc. [Accessed 20 September 2013].

[5] "OGC Standards and Supporting Documents," Open Geospatial Consortium, 1994. [Online]. Available: http://www.opengeospatial.org/standards/. [Accessed 20 September 2013].

[6] B. Spies, “Web Services Part 1: SOAP vs. REST,” May 2013. [Online]. Available: http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest. [Accessed 24 September 2013].

[7] D. Orenstein, “Application Programming Interface – Computerworld,” January 2000. [Online] Available: https://www.computerworld.com/s/article/43487/Application_Programming_Interface?pageNumber=1. [Accessed 24 September 2013].

[8] W3C, “HTTP – Hyper Text Transfer Protocol Overview,” July 2013. [Online] Available: http://www.w3.org/Protocols/. [Accessed 24 September 2013].

[9] W3C, “URIs, URLs, and URNs: Clarifications and Recommendations 1.0,” September 2001. [Online] Available: http://www.w3.org/TR/uri-clarification/. [Accessed 24 September 2013].

[10] J. Steward, “ThatGeoGuy/ENGO500,” [Online]. Available: https://github.com/ThatGeoGuy/ENGO500. [Accessed 24 September 2013].

[11] “FAQs | Raspberry Pi,” [Online]. Available: http://www.raspberrypi.org/faqs. [Accessed 22 September 2013].

[12] “Arduino – Arduino Board Uno,” [Online]. Available: http://arduino.cc/en/Main/arduinoBoardUno. [Accessed 22 September 2013].

[13] “Arduino vs. BeagleBone vs. Raspberry Pi | MAKE,” April 2013. [Online]. Available: http://makezine.com/2013/04/15/arduino-uno-vs-beaglebone-vs-raspberry-pi/. [Accessed 22 September 2013].

[14] “Sensors / Parts, Adafruit Industries,” [Online]. Available: http://www.adafruit.com/category/35. [Accessed 23 September 2013].

[15] “Sensor Web Interface for IoT SWG,” Open Geospatial Consortium, 2013. [Online]. Available: http://www.opengeospatial.org/projects/groups/sweiotswg. [Accessed 22 September 2013].

[16] “Internet of Things (IoT) and Web of Things (WoT),” Open Geospatial Consortium Network. [Online]. Available: http://www.ogcnetwork.net/IoT. [Accessed 22 September 2013].

[17] G. van Rossum, “Comparing Python to Other Languages,” 1997. [Online]. Available: http://www.python.org/doc/essays/comparisons.html. [Accessed 22 September 2013]

[18] “Arduino – Reference,” [Online]. Available: http://arduino.cc/en/Reference/HomePage. [Accessed 23 September 2013].

[19] B. Teskey, “ENGO 500 Course Outline,” September 2013. Available through Blackboard.

UNIVERSITY OF CALGARY

ENGO 500: GIS and Land Tenure #2

Literature Review: Location Aware Smart Shelves

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

DEPARTMENT OF GEOMATICS ENGINEERING

SCHULICH SCHOOL OF ENGINEERING

2013-11-18

Table of Contents

Introduction

Purpose

Information relating to customer frequency, interest, and product popularity is essential to maximizing profits and efficiency in a retail environment [1]. Analytics of this kind are useful to store owners as they make strategic marketing decisions, provided data is collected in an efficient and understandable manner. Issues that must be addressed comprise how many people come and go every day, their purchase decisions and history, and which products are sold at different times of year. Much of this data can be gathered by observing the areas of high and low customer traffic within the store, and correlating that information with product locations throughout the store, and to the objects which have been purchased. By gathering this data effectively, retail management can potentially arrange and analyze the store configuration over time, which can be an effective tool in maximizing profit and customer satisfaction [1].

In addition to knowing traffic patterns, store owners also value a well-stocked and orderly appearance. To satisfy this condition, grocery store owners will often ask their employees to *face* (or *block*) the shelves in their store's aisles. Facing a shelf involves moving all products on the shelf towards the front of the shelf (known as the face) and arranging them in a neat and orderly fashion. This allows the customers to easily see all available products and allows them to reach them without trouble as well. Therefore, keeping shelves faced is also important for grocery stores who wish to maintain their image.

This project will focus on consolidating the two above problems in the form of a “smart shelf” solution which can both track customer traffic within the store, and likewise provide a means to report on the status of an item's stock (faced or not faced). To this end, a sensor-network in the form of the *Internet of Things* (IoT) will be employed as a novel solution to these problems. Thus, the purpose of this literature review is to not only understand the perspectives and needs of store management teams, but also to study current solutions to related retail problems, and to improve or adapt these existing solutions. Upon understanding these methods and challenges, a specification for a “smart shelf” system is proposed. The proposed system will be designed to empower non-expert users from general grocers who stock shelves to upper management, to understand the needs of individual retail chains as well as develop models and data that can be applied to both the store layout, and likewise to marketing tactics, in order to maximize their profit and customer satisfaction.

Background Information

The issues stated in the introduction have been addressed before in a variety of ways. Some of these solutions fall within the field known as the Internet of Things (IoT), which is particularly well-suited to the problem. The Internet of Things is defined as a relationship between tangible objects and their virtual counterparts, achieved by equipping objects with sensors. This provides them with the ability to communicate with the web without human interaction.

Internet of Things

There exists a broad spectrum of IoT applications, many of which are proprietary. To a user, each of these objects must be controlled by their respective application or interface individually. This becomes cumbersome as the number of internet-aware devices one might use increases. Two ways to approach this problem are: to create standards which are widely accepted and can be considered as IoT protocol, similar to how HTTP transformed the web, or a more decentralized approach, where many separate, unique libraries which accommodate for the diverse number of applications

are written. With the latter approach, each object's library may be potentially written in multiple languages, with a unique API and programming style. This second approach has been adopted by an open source community called OpenRemote, which is designed to work with a variety of existing sensors already on the market. The idea is that users can design their own interface and integrate whichever sensors they want, regardless of that sensor's own protocol. Currently, OpenRemote support protocols from more than 25 different applications, such as PhilipsHue, FreeBox and Samsung Smart TV [2].

However, while this decentralized approach makes it simple to integrate various sensors for any single solution, it creates a problem whereupon programmers, hackers, and entrepreneurs must learn several different APIs for each of the devices they wish to use. Otherwise, they are limited to the sensors which a service such as OpenRemote supports. If there is a new sensor being introduced, there will be a delay before one could integrate it into an existing solution because its specific libraries would not yet be supported. This is in stark contrast to the former solution to integrating multiple sensors with the Internet of Things, where a common standard protocol can be used to access any and all devices. This means that regardless of the objects used, a single interface makes all of them commonly accessible. Such a standard has been put in draft by the Open Geospatial Consortium (OGC) [3], and is awaiting finalization as of this year. Since this method is preferred for its long-term simplicity and sustainability over solutions such as OpenRemote, this project will attempt to serve as a means to proliferate and exemplify the use of the OGC IoT standard protocol.

Existing Smart Shelf Technology

The problems stated previously, namely that of shelf facing and customer tracking, are not new problems and have been addressed in the past. Due to the requirements of our project scope and the limitations in budget, our final product will certainly be designed differently from the existing products studied below. Nevertheless, they provided insight and guidance in laying out the concepts of our design, more notably with regards to software.

Many existing solutions rely on RFID sensors [4]. RFID, or radio frequency identification, is a form of automatic identification that falls under the same category as barcode scanning and voice recognition. RFID is becoming a key technology in vast computing networks, as it provides many advantages over traditional means, such as the aforementioned barcode scanning [5]. The most common implementation of RFID involves storing a serial number on a microchip which can then transmit this number to an automatic reader via an antenna. The reader can then convert the radio waves into digital information [6]. In this way, the reader is able to gather information that is unique to each separate item that has been tagged.

Shelves equipped with RFID technology have the ability to provide information to store owners about the products which are placed on it [4]. By scanning the product, the shelf is able to identify when an item is picked up or placed, and may also track an item through the store by recording which shelf it is placed upon. Scanning an RFID chip also provides data such as the expiry date, which the shelf can store and set up a notification if the product has not been bought before that date. Furthermore, it has likewise been shown that this concept is not strictly limited to supermarket retailers, but can be expanded into other industries such as construction or site management in tracking and maintaining appropriate stock for work to be completed [7]. RFID tracking has become very popular in the world of retail, and very few alternative approaches have been used in an attempt to solve similar problems.

Another solution proposed is the NeWave Sensor [8]. This item was developed to be used primarily by store owners; however, it does not focus on improving customer satisfaction, but rather on security and monitoring of product in-store. The technology involves an antenna that can detect change within a certain area, and in this way it is able to indirectly track the items placed within that

area / on the shelf. Since it relies only on a sensory antenna, it cannot tag items like the RFID can. Its functions allow data to be collected on merchandise availability, which may indicate high or low stock or even an emergency, but cannot directly produce positional information in regards to product placement on a shelf (e.g. facing).

The advantages of sensor-based solutions can primarily be seen in the failings of these existing solutions. While RFID tags dominate in popularity, especially in regards to retail product tracking, some distinct disadvantages associated with the technique are:

1. RFID signals can be potentially noisy and may require objects to perfectly oriented with respect to the sensor in order to improve the overall accuracy [9].
2. Unlike direct sensors, or remote sensors, direct information cannot be obtained without computational overhead. For example, it was shown that the tracking the storage of construction materials required additional computation such as the gradient descent method [7]. On the other hand, weight sensors and magnetometers can detect item positions (on or off the shelf) directly, which provides a much quicker solution.
3. RFID has been in use since the 1970s [10], but it has not been widely used until recently due to its high cost. Tags are now cheaper to produce, but in a supermarket context, individual products would require a tag on every item that was to be placed on the shelf. Some companies may be able to fund this, such as in the application mentioned above [4], and in return have access to interesting, product-specific information, but the cost-barrier-to-entry may be too high for smaller competitors. Unlike RFID chips, which must be replaced per product, sensor networks provide a unique advantage in that they are a flat cost, and generally don't need to be replaced often.
4. Existing RFID based systems, in contrast to examples involving sensor networks, do not scale additively, and do not allow current solutions to be built upon and improved in an agile manner.
5. RFID tags provide little insight into customer motion throughout the store. At best they can identify product trends, but no spatial or temporal data can be garnered about customers using RFID chips, only about products themselves.

For the purposes of this project, the major concerns regarding RFID tags are numbers 2, 3, and 5 above. It is important to remember that the selling point of our project is twofold: to determine the status of shelf facing, and to provide analytics in the form of customer interest and time spent within each aisle (i.e. to determine the number of customers in a given aisle and how long they stay there). Moreover, as it would be difficult for the NeWave smart shelf to specify the position of the items on the shelf, it does not provide an efficient means to determine whether or not a shelf is faced. Finally, the NeWave solution is incapable of monitoring customer movements in detail, which makes it unsuitable for the problems we wish to solve.

Sensor networks, in contrast, offer a better solution to this problem. Unlike RFID sensors, sensor networks as used in IoT based applications can provide two primary advantages [11]:

1. Specialized sensors can be used to directly acquire information within given constraints. No additional computation is necessary for sensors to output data.
2. Sensor networks can be built in additively and are designed to scale [11], which reduces the economies-of-scale problem regarding the heavy cost of smart shelf implementations. Moreover, sensor networks provide a solution that can be additively upgraded as time progresses, reducing long-term cost as well.

For these reasons, and given that previous implementations of smart-shelves do not adequately address the problem we wish to solve, a multi-sensor approach is preferred for the implementation of our desired solution.

Retail Analytics

Currently, there are two main ways in which retailers collect data about traffic in stores: using video or using cell phone signals. These two ways have been researched and implemented because it typically requires little additional hardware setup. Many stores already have an existing set of closed circuit television (CCTV) cameras in their stores, and nowadays most customers carry cell phones with them while they shop. Leveraging the existing technology is a common strategy employed by several commercial companies and research groups. Some specific examples of both cases are discussed here.

As mentioned, many stores are already equipped video cameras throughout the store, in order to track suspicious behaviour and have resources for offline investigation if any theft occurs. However, having people monitoring surveillance cameras can be quite inefficient, as it requires a lot of time and some events (particularly in larger facilities) could go unnoticed. Hence, having some kind of automatic detection or monitoring systems can be useful. As developed and tested by Senior et al. [12], the video recordings can be augmented with various algorithms to help real-time monitoring of stores. For example, people can be tracked as they enter or leave the store, including what they carry with them when they enter. See Figure 1 below:

Figure 1: Colour density heat map of customer activity [12]

This is useful information to help with returns fraud, where people take new items and return them without having actually purchased them. In Popa et al. [13], video cameras are used to understand the shopping behaviour of customers. For example, they use various detection algorithms to determine if a shopper should be classified as goal-oriented or disoriented based on their walking speed. In addition, they detected facial expressions and interactions with objects.

Some commercial solutions are also available for gathering and communicating retail analytics. One such service is called ShopperTrak, which offers a variety of solutions for retailers to understand key questions about their consumers. Such questions include, “is my marketing effective?” and “which zones attract the most traffic?” [14]. This service again leverages video camera technology, which is often undesirable due to considerable image processing overhead, and uses mobile analytics from cellular phones, which creates privacy concerns amongst consumers [15].

Another commercial solution is Path Intelligence, which also has a variety services such as location usage analysis using interactive heat maps, signage effectiveness reports, flow management analysis and dwell time reports. All of these services help retailers understand their customers better. Path Intelligence uses the strength of cell phone signals in order to triangulate a location for each

customer. While the locations are known, the identity of each consumer remains anonymous [16]. An example of Path Intelligence can be seen in Figure 2 below, which shows how customers are tracked as they move throughout aisles within a store:

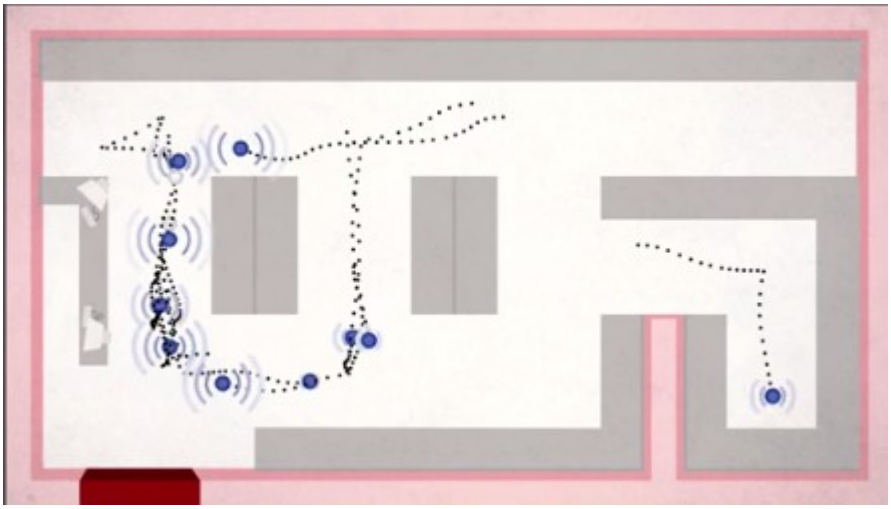

Figure 2: Illustration of path mapping using Path Intelligence [16]

Other retail analytics companies also use cell phones in order to track shopper movements, but they use MAC addresses instead of cell phone signals sent to cell towers. If a phone is Bluetooth or WiFi-enabled, it will broadcast its own MAC address in order to communicate with other devices. Based on detection of MAC addresses belonging to each device, their locations can be determined. Aggregate reports formed from this data provide insight into current waiting times at cash registers, optimize store location, and consumer shopping patterns [17].

Sensor Servers

In order to be able to communicate with different sensors simultaneously, a sensor server is typically used. There are several types of existing sensor servers which have been marketed commercially. Some of them are only able to interface with certain sensors, such as the PCW-SSRX Sensor Server made by SenSource [18]. This server communicates specifically with proprietary wireless sensors equipped with radio frequency transmitters, or wired sensors. From one or more servers, information can be communicated to a central PC, which can query the server using proprietary Server Manager Software or by simple ASCII commands.

Another sensor server has been created by Oracle, which is well-known for its database management systems. The Oracle Sensor Edge Server integrates data from different sensors, which includes RFID, temperature sensors, humidity sensors and others, as well as command/response-type equipment. It is the middle-tier component which communicates between these objects and applications [19].

Project Objectives

Aisle Management

Aisle management, a term which means changing the placement of products to increase traffic, sales, and profits is one method that stores use to succeed in the competitive retail market. According to Larson [20], the value of moving a product's shelf space can be estimated by tracking customer traffic within a store. Working with this higher level statistic (treating the aisle as an entity) as opposed to a categorical approach, resource costs for analysis can be reduced. Dreze et al. [21] found that on average, a 4-6% sales increase could be found by optimizing product placement.

Moving a product from the least ideal to the most ideal location could have an increase in sales of up to 60 percent. Additionally, creating a layout in which customers browse more isles is important to creating sales. A study by Coca-Cola [22] found shoppers travel and average of 41% of the aisles on a typical shopping trip. These findings show that tracking customer traffic will be a valuable resource in optimizing store layout.

Focusing on aisle management can be beneficial for improving a store's success. When products are grouped into different zones (aisles), these aisles can be monitored and analyzed in order to try to maximize the traffic in and profits from each of the zones. It was also found that if a greater number of people observed a certain product display, traffic increased in the aisle which had that product. Aisle length also has an impact, as it has been noted that consumers are less likely to go down an aisle that is too short or too long. While there are a number of principles which can be followed (for example, grouping products by type is more effective than grouping them alphabetically by their name) [20], stores may also find that some situations or trends are specific to the demographics and organization characteristic of their own store.

Further research, conducted by P. Chandon et al. (2009) [23], used eye tracking technology to measure attention towards items on supermarket shelves. The technology measured eye movements such as fixations and jumps. In complex settings, such as a supermarket shelf, eye fixations represent object identification, and the movement is a good indicator of the direction that the eye is looking. This study suggests that not all shelf locations attract equal attention, and that the items located near the center of two shelf displays were noted more often [23]. It is inferred from this that an increase in shelf space, or an increased number of facings, draws more attention. The report concludes that the effects of increasing the facing of the display area directly and positively impacts the profit made by the store, making it an important aspect for all store owners to consider.

Shelf Facing

The facing of a product refers to the location and amount of shelf space is allocated to it. According to research by Chandon et al. [23], eye movement studies, shopper surveys, and field experiments have all confirmed that large increases in shelf space increase sales even when the price and location of the product stay the same. A logical extension of this is that a product's facing will not be optimal after stocks begin to be depleted by shoppers and the product does not fill the front of the shelf properly. To combat this, a common task for store clerks is to pull stock from the back of the shelf forward to create the illusion that the shelf is full. This is essentially bringing the facing of the product back to its allocated space. By tracking which products need to be faced, management and store owners can see areas which need attention at a glance without needing to travel the store personally. This data would also indirectly show popularity based on the number of times that it needs to be stocked, and out of stock problems if a clerk had faced an entire aisle but some products are shown as missing by the sensors. An example of shelf facing can be seen in Figure 3 below:

Figure 3: Shelf facing of different sizes according to brand [23]

Such data could be useful in other applications as well, such as one that makes use of detailed models to estimate stock levels. These statistical methods use historic and current data such as order/delivery numbers, sales averages, volatility, shelf space and point of sale monitoring [24]. This type of approach has been put to use in KSS Retail's Heartbeat [25].

Stores often arrange their stock as specified in a planogram, which acts as a map for their products on their shelves. However, planning out how to arrange the products is one matter – ensuring that products are in the correct locations and maintain an organized appearance is another matter. This task is often very labour-intensive and time-consuming. In order to address this problem, Intel has developed an intelligent shelf compliance solution. The solution is a robot named AndyVision, which navigates the aisles of the store using sonar technology and uses a Microsoft Kinect as eyes to check the shelves [26]. The collected images can wirelessly alert staff when stock is low and if items are disorganized or misplaced. These images, once processed, can also help staff identify product placement issues (e.g. if a product has been allocated too much or too little space). While the research project prototype was a robot, a typical retail deployment might consider using existing cameras or installing miniature cameras on the sides of shopping carts, as a robot might inconvenience shoppers during opening hours [26]. Unfortunately, both the cost and computational complexity of computer-vision based solutions do not fit into the desired model we wish to create.

The main goal of monitoring shelf facing is to be able to alert staff as to whether or not a shelf is faced, so that if it is not, an employee can remedy the situation. In order to reduce some of this time-consuming work, stores sometimes use shelves which have tracks, dividers and springs to automatically organize products, pushing them to the front of the shelf. These are available in different sizes to accommodate different types of stock [27]. An example of such a system can be seen in Figure 4, which shows a self-facing shelf using such a product:

Figure 4: Example of automatically-faced shelves [27]

Such systems, while automatically facing shelf products, do not work if there is no product in stock.

In such a scenario, one proposed solution is to integrate sensors in order to make the object *aware* of such situations. Such shelves simplify the facing problem significantly as they reduce the problem from “is the shelf faced?” to “does the shelf have any product at all?” This is important as it can significantly save costs in developing a solution to this problem.

Hardware Specifications

To address both aforementioned goals of this project, multiple sensors will be used in an IoT style. The sensors require a connection to a micro-controller, such as the Netduino [28], Arduino [29], or a RaspberryPi [30]. This is a necessary part of the project, as the multi-sensor network we wish to create to meet our objectives needs some way to interface with the internet, in order to join the Internet of Things. A summary of the technical specifications for these boards can be found in Tables 1 through 3 below:

Table 1: Netduino technical specifications [28]

Parameter	Value
Processor	Atmel 32-bit micro-controller
Speed	48 MHz, ARM7
Storage	128 kB
Memory	60 kB
Network	Ethernet
Other	Requires .NET Framework (Microsoft)

Table 2: Arduino technical specifications [29]

Parameter	Value
Processor	ATmega328 8-bit micro-controller
Speed	16 MHz
Memory (Including Storage)	32 kB
Network	No
Other	Requires additional components for internet connectivity

Table 3: RaspberryPi technical specifications [30]

Parameter	Value
Processor	Broadcom BCM2835 32-bit processor
Speed	700 MHz, ARM7
Storage	~4GB depending on SD card and expandable storage (recommended storage reqs.)
Memory	256 or 512 MiB
Network	Ethernet, supports USB Wifi
Other	Can support multiple programming languages, and a full GNU/Linux operating system.

Because of the computing requirements of the project, having a large number of sensors interfaced to one microcontroller, and the disparity in ability among boards, the Raspberry Pi stands out above the rest for this application. Moreover, the SensorWeb Lab at the University of Calgary owns a couple of these boards, which makes them ready and available for prototyping.

One of the options considered for this project's implementation of a smart shelf are the aforementioned spring load racks (see Figure 4). Spring loads (also known as product pusher racks or shelf facing racks) are shelving racks that are spring loaded and they automatically push the product towards the front of the display, guaranteeing a full, well organized shelving system [31].

Advantages for using spring loads (product pusher racks) are [31]:

- Maintains consistent facing and organization of products
- Guarantees optimal use of shelf space and inventory
- Increases sales potential at the point of merchandise

For this project, we likewise want to use a multi-sensor setup that can interface with our micro-controller board in order to make our sensor network “smart” or “aware.” Some proposed sensors include light sensors, range sensors, and magnetometers amongst others. While the exact sensors required to solve the problem have not yet been decided upon, the following specifications are necessary for the project to reach fruition.

Functional Specifications

The following functional specifications need to be met in order to deliver a product that allows our sensor network to collect data unobtrusively with respect to regular store activities:

1. The proposed shelf should be able to tell when a product is no longer faced. For example, in the case of spring load racks, magnetometers could be attached to the back of the spring loaded wall of the rack, and magnets at the face of the shelf could be fixed such that the magnetometers reach specific readings when the shelf is unfaced.
2. The proposed shelf should be able to determine if somebody is standing in-between the length-wise area of the shelves, such that it is possible to know whether a customer is standing between the two edges of the shelf-space. This makes it possible to know where in an aisle a customer might be. An example might be to use range or motion sensors at each edge of the shelf. If a customer crosses one side but not the other, this could indicate that they are in between both sensors.
3. The shelf should be sized similarly to existing store shelves, and should not provide any lower limits on the amount of stock that can be placed on a shelf. In this same way, the shelf should not obstruct or otherwise place limits on the amount of space between aisles compared to that of existing shelves.
4. The final device should not break without internet access, but should expect reliable internet infrastructure in place in order to perform.

Non-functional Specifications

The following non-functional specifications should be met in order for the hardware to perform as intended:

1. The final product should conform to the Open Geospatial Consortium standard for IoT SWG [3]. Specifically, the data output from the hardware should be

interoperable with a conformant database built from this standard.

2. Relatively stable internet access should be available (either through a wired or wireless connection) so that the shelf can update its status as frequently as possible.
3. Power to the shelf should be made available. For a single shelf consisting of one Raspberry Pi controller, a 700 mA (5V) power source is necessary to operate [30]

Software Specifications

For any project involving software, it is important to identify our goals and to define the specifications in advance. As mentioned in previous sections, we found that there is a clear advantage for store owners and retailers who wish to know more about how their store is faced, as well as the distribution and pattern of customer positions within the retail outlet.

Functional Specifications

The following functional specifications need to be met in order to deliver a product that allows non-professionals to interact with our proposed system:

1. The ability to view shelf facings in real time, and determine where products in the store need to be restocked.
2. The ability to track customers entering and exiting each respective aisle, and determine how long each customer stayed in that section of the store.
3. The ability to generate some basic analytics based on data from the sensors. This can include items such as determining hotspots or busy times around the store, analyzing which parts of the store are more popular (most shelves empty), and other organizational information, such as busy times within the store, or average time left unfaced.

These specifications, if achieved, will result in a product that produces value for non-professionals in a managerial or aisle-management context within a retail organization.

Non-functional Specifications

Although there are many things to consider when discussing the overall functionality of our end software, it is important to also note some of the non-functional requirements, as follows:

1. The system should be painless and simple to introduce. Specifically, it should not require the user to do anything beyond setting up the hardware. For this reason, a central website that distributes management of multiple entities is preferred to a more decentralized system, where entities would set up their own specialized hardware for their application.
2. The system should be secure and prevent unauthorized entry. This is to protect consumers from privacy concerns, as well as to protect both managers and the data server from malicious entities.
3. The system should be robust enough that events such as power loss or switching products between shelves should not produce any significant overhead to the system.
4. The system should be easy to manage and should provide managers with the ability to readily pull up any information from a given store or branch that they might choose.
5. The system should have received an extensive amount of testing, preferably upwards

of 30 hours of actual use testing and bug-fixing. As the project expands the test requirements will also be expanded to maximize ease-of-use of the final product.

6. All of the software should operate using the OGC IoT SWG standard [3], and should be published as free, open-source software for anyone to access and improve upon.

These specifications are particularly necessary in that it will be important to establish ease-of-use in order to market the final product. If the system is painless and simple to introduce, there is a more likely chance that such a system will establish itself in the market. This is often referred to as the technological acceptance model [32], and makes a strong argument for ease-of-use over initial functionality. Specifically, it argues that system adoption is more likely when systems are designed in a fashion that emphasizes ease-of-use over sheer feature count. This also factors in to our 3rd, 4th, and 5th non-functional specifications, because we find that they represent a measure of ease-of-use in slightly different ways. Therefore, in order to follow the technological acceptance model, the initial product will focus more strictly on the ease-of-use problem before regarding feature addition.

With regards to information security, the website will enforce a strict client-server authentication model, where specific shelves will be attached to different accounts based on universally unique identifiers (UUIDs). The specific research for this is not fleshed out here, as many common development frameworks are bundled with a sufficient client-authentication system, and will be largely implementation dependent. However, it should be noted that for the majority of the website development proposed by this project, it is likely that client-side computations will be necessary, as the group has not been able to secure any hardware with the capability of performing server-side computations and storage. Therefore, modern technologies such as HTML5, CSS, and Javascript will be necessary in order to allow us to perform computation of database results and analytics, as well as allow us to use some form of local storage for result caching.

Methodology

To accomplish the two metric tracking goals detailed previously the group will be split into two teams. The first team will focus on the hardware configuration and sensor interfacing while the second team will focus on developing the necessary software and website.

Moreover, the team has chosen an incremental approach in order to create a product capable of shelf face detection and customer-traffic tracking. This type of development is a combination of waterfall and iterative development methodologies [33]. These methodologies are combined by adding an iterative prototyping period after the initial investigation and requirements definition are completed in the waterfall fashion. An example of this type of work-flow can be seen in Figure 5, below:

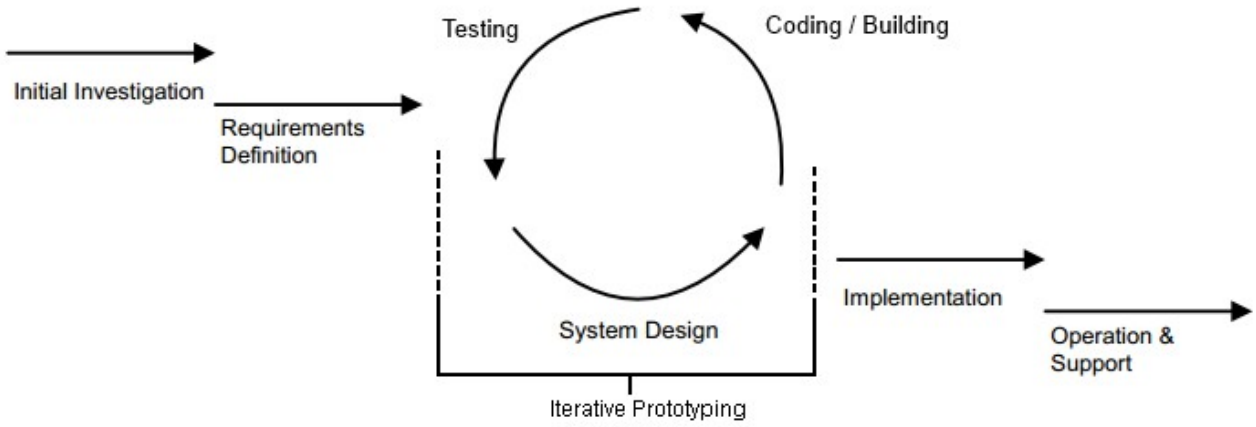

Figure 5: Flowchart diagram of iterative development process / work-flow (edited) [33]

This strengths of this type of development are [33]:

- The project is maintained with written documentation requiring approval/sign-off at designated milestones.
- Stakeholders can be given evidence of progress
- Gradual implementation allows evaluation of each incremental change
- Well suited to projects where requirements may change due to new knowledge, technology or expectations
- Well suited to leading-edge applications

However, despite the many advantages listed above, there still exist possible pitfalls which teams should be aware of so that appropriate measures can be taken to avoid them. Examples of such drawbacks with this methodology include:

- Difficult components may be ignored until late in the project time-frame
- Well defined interfaces are crucial since not all components of the project will progress at the same rate

Using the above team-based iterative methodologies, each of the two teams will attempt to complete the responsibilities outlined in the following sections.

Hardware based methodology

The hardware and interfacing team's responsibilities can be broken down into to the following tangible tasks:

1. Determine which sensors & micro-controller are most appropriate for the chosen specifications
2. Design the proposed sensor configuration & initial testing
3. Develop interface to send data to the server
4. Construction of prototype
5. Testing of prototype
6. Improvement to prototype (interfacing or hardware configuration)

Task 1: Determination of sensors & micro-controller

The sensor array being developed for this application is focused on embracing the IoT mantra as a main design influence. This means that the sensor network should include a number of sensors that are networked and accessible in a way that allows a full interoperability of interconnected devices..., providing them with an always higher degree of smartness by enabling their adaptation and autonomous behavior, while guaranteeing trust, privacy, and security" [34]. The required sensors will be decided upon based on IoT values, OGC standards, the technical specifications detailed by the team, and the goals of increasing sales in a retail environment.

The Sensor Web Research group at the University of Calgary [3] will be the first point of entry into deciding what sensors and micro-controllers will be used after a period of hands on experimentation with their resources. Some of these resources may be leveraged for the duration of the project. It is likely that not all of the required components for the project will be available in which case these will need to be acquired via an online vendor. A broad set of sensors may be tested for an initial investigation before the sensors for the final configuration is ordered.

Task 2: Design the proposed sensor configuration & initial testing

With the technical specifications in place, a sensor configuration will be developed to meet these design requirements. The main factors to be considered are cost, effectiveness, durability, robustness, and the protection of privacy.

To meet the cost requirements, different arrangements of sensors should be examined to find the best configuration of sensors and micro-controllers. With the cost of a micro-controller board being as much or more than most sensors, a design should emphasize on providing the needed detection methods for as much area as possible per micro-controller. Varying costs of sensors such as infrared ranging, magnetometers, luminance sensors, etc, may allow different combinations to achieve the same end goal.

To be effective, sensors need to be mounted in the optimal location. Logical mounting points for customer traffic tracking depend on the type of sensor used. A more naïve solution would be to track movement past certain gates, in which case sensors at the entrances and exits to aisles would count how many customers travel through the aisle, and potentially time spent in an aisle. Adding additional sensors in the middle of the aisle could also be beneficial to increase the resolution of the statistics gathered. Another option is image based soft-biometrics, which is more expensive both in terms of price and computationally. This type of tracking uses descriptors such as height and clothing color to track people in images, even between separate cameras [1]. Data from either method will be useful in correlating the length of the aisle, the products contained in it, and the time spent in a section. In terms of facing detection, Christenfeld [23] found that when identical products are found side by side (an allocated product facing), that consumers tend to choose the middle product. This means that for products with greater than one facing, the sensors should be located in a position that monitors the middle of the display.

Durability is a great concern with any business purchase, so the sensors must be configured in a way that is not susceptible to major wear from normal customer behavior. This durability could be the result of mounting sensors in a way that customers will not interact with them, or in a way that the sensors are protected.

The system must be robust, so that the system can stay productive over long periods of time. In the software dealing with the sensors, functions to test and calibrate the sensors may be needed to detect and remedy errors in the system. The type of tests that each sensor will be subject to depends on the sensors used. Moreover, as with any IoT implementation, privacy is of primary concern to consumers [15][34]. The customer traffic tracking aspect of the project is the most likely to create privacy issues. To protect consumer privacy, the system will need to be developed in such a way that people are not identifiable from the data collected (i.e. data is anonymized in some fashion). Additionally, the data collected will only be available to authorized users, and should remain out of the public forum.

With the sensor design and set-up tentatively chosen, an intermediate step must be taken to verify that the product will work as envisioned. This will require the sensors to be connected to the micro-controller and data will need to be accessed to ensure that the intended behaviour is performed. At this stage the data read from sensors does not necessarily need to be transformed, analyzed, saved, or sent anywhere. Verifying that the sensor is giving the correct output will be a large but necessary step in the design process. This will require programming to read the output of the sensors as well as circuitry design.

The final part of this task is drawing up schematics for the system as it would be installed on a retail shelf based on the knowledge gained through previously completed steps. This design will be a recommendation which can be adapted as new challenges arise. The format of this design should be a technical drawing or 3D model to ensure that the envisioned product can be created in 3D space.

This may highlight some flaws in design choice which can be changed or adapted before sinking significant development time into interfacing sensors or building a prototype.

Task 3: Develop interface to send data to server

Once the configuration of the sensors has been chosen, the next step will be programming the software that handles the data. This will consist of the following steps: reading, interpreting, and temporarily storing data in a robust way; developing an interface between the server and micro-controller; testing and verification.

Depending on the sensors used, the data may come in analog or digital format. Useful data will be streaming from multiple sensors concurrently, and along with other metrics such as time of day, will need to be sent to the database. Depending on the timing of incoming data, temporary storage will need to be allocated and managed. Due to the relatively low computing power of some micro-controllers, memory management will be an important consideration during this step.

Collected data must be packaged and sent in a way that is useful and efficient for transferring as well as storing in the database supplied by the Sensor Web Interface for IoT Standard working group [3]. The complexity of this task will very much depend on the micro-controller used and its supported languages. Although the prototype will see a very limited amount of data transfer in practice, the design should be created with a much larger scale of transactions in mind.

Once the interface has been developed, testing of the system can begin. This testing will ensure that the system can meet technical specifications before being built into a more permanent prototype. Testing will focus on correct results for test cases, uptime for the micro-controller, and reliability of the data being sent to the database. If any significant problems are encountered, they will be resolved before creating the system prototype.

Task 4: Construction of prototype

Once the product has a set of working sensors with a database connection, construction of the prototype can begin. This will be either a purpose built shelving unit or an adaptation of an existing shelving unit as described in the design completed in task 2. Sensors will be mounted and connected to the micro-controller, which will need its internet and power supplies implemented. Any non-permanent durability enhancements can be done at this point as it will be beneficial to test these adaptations.

Task 5: Testing of prototype

Having constructed the prototype, testing will commence. To begin, testing will be done against known cases which will have been developed as part of tasks 2 and 3. To supplement this, testing will also be done to verify the system performance in a real life scenario, which will involve team members adjusting stock over long periods of time. Data collected during this testing session could potentially be used for presentation purposes so it will be organized and saved. If bugs or improvements are encountered, they will also be recorded.

Task 6: Improvement of prototype

Any potential improvements or bug fixes derived from prototype testing can be added after they are verified by the team. Any changes to the hardware or software will have to be regression tested to ensure new bugs are not introduced and that the system is still functional. This will be an iterative process which will run until the project has reached completion.

Software based methodology

Collecting data from the sensors alone is not enough to achieve the goals of the project. The team also aims to interpret and present this data by means of a website. The software and website team's tasks are as follows:

1. Develop interface between website and database
2. Testing of database interoperability
3. Design website prototype
4. Testing of website prototype
5. Improvements to website prototype and website/database interface

Task 1: Develop interface between website and database

Similar to the hardware team's interfacing of the micro-controller to the server, the software team will need to interface the website and database. This should be done with security, efficiency, and robustness in mind. This task will require a small amount of website development in order to verify that the interface is working properly, but at this stage minimal time should be invested in an appealing graphical user interface (GUI) or layout issues.

Task 2: Testing of database interoperability

Once the interface has been developed, testing should be conducted to verify that the correct queries are being made, that errors can be identified, and that the interface is reliable. This task may be done in conjunction with task 4, after most of the website development is done.

Task 3: Design of website

Designing the website will be the major piece of work tackled by the software and website team. To begin with, the website framework must be chosen. Additionally, decisions pertaining to what browsers to support, what authentication model to use, update period, hosting solutions, and if the connection should be encrypted also need to be decided. Once the basic infrastructure has been planned, the team will focus on what kind of information is valuable to users, and what the best way to present this data is. This will cover things such as what statistics can be calculated, what kind of graphics can be generated to represent these data/statistics, what kind of navigation and structure should the website use, all the while maintaining a system that is easy to expand if new functionality is required. Information that can be derived from the data collected could include things such as stagnant products, traffic to aisle facing condition, historical vs. current trends, or a variety of others. The website will be the most visible part of the project, so the experience must be polished and easy to understand.

Task 4: Testing of website

The website should be tested on all supported browsers using a set of test cases as well as simulating real life conditions. Data and generated statistics need to be checked for accuracy as well as proper display. All navigation should be verified and authorization needs to be tested to ensure that user's data is secure. Feedback should also be gathered from user's outside the team to find potential improvements.

Task 5: Improvement of website and website/database interface

Any bugs or improvements noted during the testing tasks will be considered and implemented if the

team agrees they are achievable and worthwhile. Outside opinions will also be considered. Again this will be an iterative process of updating and testing until the completion of the project.

Conclusion

In this document, various concepts and implementations regarding a smart-shelf product were explored. Compared to traditional methods involving RFID tracking, and alternative methods using custom solutions, it was determined that to solve the problems of shelf-facing and customer-traffic tracking, a “smart shelf” with a multi-sensor network built in is preferred, for both the scalability of sensor networks [11], and the speed and ease that direct acquisition of sensor readings provide. The proposed smart-shelf product will likewise be connected to the Internet of Things, which unlike some alternative implementations, will provide management and supermarket workers the ability to react flexibly to the demand of the store in real-time.

In addition to examining the benefits of the proposed system over current alternative implementations, both a hardware and software specification were laid out for the remainder of the project. In particular, it was decided that due to its power, versatility, and availability, the Raspberry Pi board is preferred to alternative micro-controllers proposed for the project. Lastly, a simple web-interface with an emphasis on ease-of-use as per the technological acceptance model was laid out for later implementation. Finally, a task model for the remainder of the project lifespan was projected, with an emphasis on iterative, agile development. Such a cycle encourages active testing during development with an emphasis on stability and a working product.

References

[1] S. Denman, A. Bialkowski, C. Fooks, and S. Sridharan, “Identifying Customer Behaviour and Dwell Time using Soft Biometrics,” *Video Anal. Bus. Intell.*, vol. 409, pp. 199–238, 2012.

[2] OpenRemote, “OpenRemote: Open Source for Internet of Things,” *OpenRemote: Open Source for Internet of Things*, 2012. [Online]. Available: http://openremote.com. [Accessed: 14-Oct-2013].

[3] Open Geospatial Consortium, “Sensor Web Interface for IoT SWG,” *Sensor Web Interface for IoT SWG*, 2013. [Online]. Available: http://www.opengeospatial.org/projects/groups/sweiotswg. [Accessed: 22-Sep-2013].

[4] Sini Syrjala, “RFID Arena,” *RFID Arena*, 13-Sep-2012. [Online]. Available: http://rfidarena.com/2012/9/13/%E2%80%9Csmart-shelves%E2%80%9D-the-store-shelf-of-the-future.aspx. [Accessed: 19-Oct-2013].

[5] Peter Russer and Uwe Siart, *Time Domain Methods in Electrodynamics*. Berlin Heidelberg: Springer, 2008.

[6] RFID Journal LLC, “RFID Journal FAQ 7638,” *RFID Journal FAQ*, 2013. [Online]. Available: http://www.rfidjournal.com/site/faqs#Anchor-How-7638. [Accessed: 12-Nov-2013].

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

[8] NeWave Solutions Ltd., “NeWave,” *NeWave*, 2013. [Online]. Available: http://newavesensors.com/products/smartshelf. [Accessed: 19-Oct-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] RFID Journal LLC, “RFID Journal FAQ 36680,” *RFID Journal FAQ*, 2013. [Online]. Available: http://www.rfidjournal.com/site/faqs#Anchor-If-36680. [Accessed: 12-Nov-2013].

[11] Steve H. Liang, Arie Croitoru, and C. Vincent Tao, “A distributed geospatial infrastructure for Sensor Web,” *Comput. Geosci.*, vol. 31, pp. 221–231, 2005.

[12] A.W. Senior, L. Brown, A. Hampapur, C.-F. Shu, Y Zhai, R.S. Feris, Y-L. Tian, S. Borger, and C. Carlson, “Video analytics for retail,” in *Video analytics for retail*, 2007, pp. 423–428.

[13] Mirela Popa, Leon Rothkrantz, Zhenke Yang, Pascal Wiggers, Ralph Braspenning, and Caifeng Shan, “Analysis of Shopping Behavior based on Surveillance System,” in *Analysis of Shopping Behavior based on Surveillance System*, 2010, pp. 2512–2519.

[14] ShopperTrak, “ShopperTrak Solutions,” *ShopperTrak Solutions*, 2013. [Online]. Available: http://www.shoppertrak.com/products. [Accessed: 01-Nov-2013].

[15] Katherine Poythress, “Is it OK for retailers to track shoppers with smartphones?,” 06-Aug-2013. [Online]. Available: http://www.utsandiego.com/news/2013/aug/06/retailers-tracking-shoppers-privacy-concerns/. [Accessed: 17-Nov-2013].

[16] “Path Intelligence Ltd. Technology - Path Intelligence,” *Path Intelligence Ltd. Technology - Path Intelligence*, 2013. [Online]. Available: http://www.pathintelligence.com/technology/. [Accessed: 01-Nov-2013].

[17] Future of Privacy Forum, “Mobile Location Analytics: Code of Conduct,” *Mobile Location Analytics: Code of Conduct*, 22-Oct-2013. [Online]. Available: http://www.futureofprivacy.org/wp-

content/uploads/10.22.13-FINAL-MLA-Code.pdf. [Accessed: 01-Nov-2013].

[18] Sensource, “Sensor Server for Direct-to-PC or Network Integration,” *Sensor Server for Direct-to-PC or Network Integration*, 2013. [Online]. Available: http://www.sensourceinc.com/PCW-SSRX-Sensor-Servers.htm. [Accessed: 09-Nov-2013].

[19] Oracle, “Oracle Sensor Edge Server,” *Oracle Sensor Edge Server*, 06-Aug-2005. [Online]. Available: http://docs.oracle.com/cd/B14099_19/wireless.1012/b13819/rfid.htm. [Accessed: 09-Nov-2013].

[20] Ron Larsen, “Core Principles for Supermarket Aisle Management,” *J. Food Distrib. Res.*, vol. 37, pp. 107–111, 2006.

[21] Xavier Dreze, Stephen J. Hotch, and Mary E. Purk, “Shelf Management and Space Elasticity,” *J. Retail.*, vol. 70, no. 4, pp. 301–326, 1994.

[22] *Study Finds that Consumers Shop Less than Half the Store*, 4th ed., vol. 15. POPAI News, 1991.

[23] Pierre Chandon, J. Wesley Hutchinson, Eric T. Bradlow, and Scott H. Young, “Does In-Store Marketing Work? Effects of the Number and Position of Shelf Facings on Brand Attention and Evaluation at the Point of Purchase,” *J. Mark.*, vol. 73, pp. 1–17, 2009.

[24] Dimitrios A. Papakiriakopoulos, “Automatic Detection of Out-Of-Shelf Products in the Retail Sector Supply Chain,” Athens University of Economics and Business.

[25] Dunnhumby, “KSS Retail Heartbeat,” *KSS Retail Heartbeat*, 2013. [Online]. Available: http://kssretail.com/solutions/out-of-stock-detection/. [Accessed: 07-Nov-2013].

[26] Intel, “Intelligent Shelf Compliance Solution,” presented at the National Retail Federation Conference and Expo, New York City, MA, USA, 2013.

[27] 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].

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

[29] Arduino, “Arduino,” *Arduino*. [Online]. Available: http://arduino.cc/. [Accessed: 10-Nov-2013].

[30] 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].

[31] Full Steam Marketing & Design, “Merchandising / Spring Load,” *Merchandising / Spring Load*. [Online]. Available: http://www.fullsteam.com/merchandising/spring_loads. [Accessed: 11-Nov-2013].

[32] Albert L. Lederer, Donna J. Maupin, Mark P. Sena, and Youlong Zhuang, “Technological Acceptance Model and the World Wide Web,” *Decis. Support Syst.*, vol. 29, pp. 269–281, 2000.

[33] Department of Health & Human Services - USA, “Selecting a Development Approach.” Office of Information Services, 27-Mar-2008.

[34] Luigi Atzori, Antonio Iera, and Giacomo Morabito, “The Internet of Things: A survey,” *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, 2010.

UNIVERSITY OF CALGARY

ENGO 500: GIS and Land Tenure #2

Technical Deliverables Report

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

DEPARTMENT OF GEOMATICS ENGINEERING

SCHULICH SCHOOL OF ENGINEERING

12/06/2013

Table of Contents

List of Figures

Index of Tables

Introduction

Purpose & Scope

The purpose of this document is to summarize and describe the progress of group *GIS & Land Tenure #2*, in particular to report the development of an open-source Internet-of-Things initiative. This report describes in detail the work done by the group with regards to the planning process of the project, the development of a prototype smart-shelf as part of the project design, and the development of a user-friendly web-based interface with which to interact with our shelf prototype.

Upon completion of describing the work done thus far in the project lifetime, the project schedule and timeline is evaluated. Milestones accomplished and problems encountered are discussed and evaluated, and finally a risk-assessment for the future of the project lifespan is considered. A final summary of the project status is stated, followed by a brief conclusion on the work done and goals to strive for in the remaining life of the project.

Project Overview

The focus for this project is about the concept of “Internet of Things”. The Internet of Things (IoT) is a scenario in which objects, animals or people are provided with unique identifiers and the ability to transfer the data automatically over a network without requiring human-to-human or human-to-computer interaction [1]. The Internet of Things, while barely known within the common vernacular, can make data associated with everyday objects easily accessible and meaningful, which will have a profound impact on our way of life.

The number of IoT applications are expected to rapidly increase in the future, and likewise drive tremendous value for both businesses and people [2]. However, a budding problem within the IoT ecosystem is that for any given application, we often find that there are implementation specific methods. Similar to how HTTP unified the web, part of this project focuses on implementing an Internet of Things application in a standards-compliant fashion, to reduce the amount of intellectual overhead for developers who wish to learn how to interact and develop the IoT platform, as well as provide users an interoperable gateway for IoT devices that they can trust.

To this end, our primary objective for this project is to develop an application that applies Internet of Things concepts in order to serve as an example for future professionals and developers to create location-aware services, and likewise to assist in pushing forward the IoT Sensor Web Group (SWG) standard [3] proposed by the Open Geospatial Consortium (OGC). OGC standards are technical documents that specify interfaces and/or encodings. Software developers use these documents in order to build open, accessible interfaces and encodings into their products and services [4]. Given these objectives for our project, two explicit requirements were set:

1. Develop a hardware prototype of some sensor setup that uses the Internet of things, and interface it with some web-based software system using the OGC IoT SWG standard [3].
2. Develop this application as Free, Open-Source Software (FOSS), such that anybody can access the code and tweak, develop, mashup, and redistribute it past the lifespan of the project.

From these, our group was given freedom to plan and manage what kind of project we wished to develop.

Project Conceptualization and Planning

Given the above stated goals of our project, our team was given free-reign to decide what type of

solution we believed would be a good fit to act as both a proof-of-concept and working implementation of an Internet of Things application. It was important to consider that our designed project would not only act as an example case for the upcoming OGC IoT standard, but would likewise have impact, and hopefully be useful in some aspect.

That said, much time was devoted to determining an appropriate application for an IoT-based solution. For the majority of the first six to eight weeks of development, serious consideration went into specifically defining and refining our project scope and definition. Most of this progress has been consolidated into an appropriate literature review [5], which likewise defined both our prototype and web-interface specifications. Additionally, it demonstrates that our project would have impact and provide value to store clerks and managers who would be the primary target of our developed solution [6] [7].

To meet these objectives, we chose to develop a smart shelf system. Such a system will be able to track customer movements throughout the store, as well as determine the amount of stock / facing of the store shelves. A more detailed description of this system can be found in the *Prototype Development* section below. Based on this, our group divided the project into two major categories: Our shelf prototype development, and the development of the web-based user interface (website). Overall, these two facets of our project are connected through the use of the OGC IoT SensorThings API as seen in Figure 1:

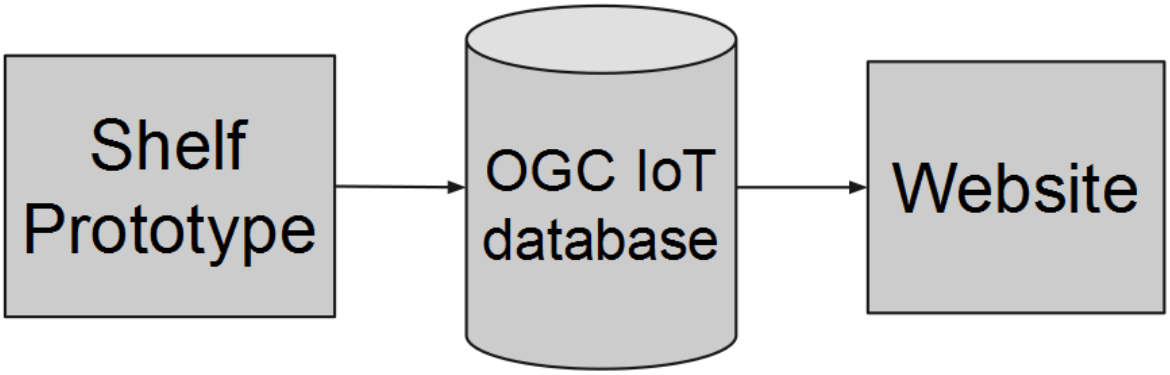

Figure 1: Division of major project components

For each of these project components, a use case analysis was conducted. A use case study is done to define interactions between a system and an actor or actors to achieve a goal. This is a high level-planning tool to help develop requirements. It was chosen as a first step in development due to its precedence in modern software engineering as well as its ability to evolve with the project.

The first step taken was to create a use-case or class-interaction diagram. Creating a visual overview of the system helped to define what functionality the final system would support. Figure 2 shows this Use-Case diagram for the Shelf Prototype:

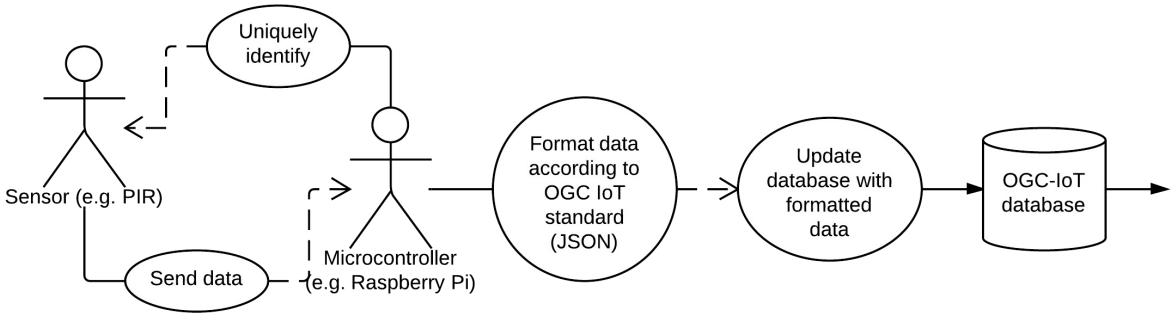

Figure 2: Use-case diagram for sensor prototype

The corresponding use-case diagram for our website can be seen in Figure 3 below:

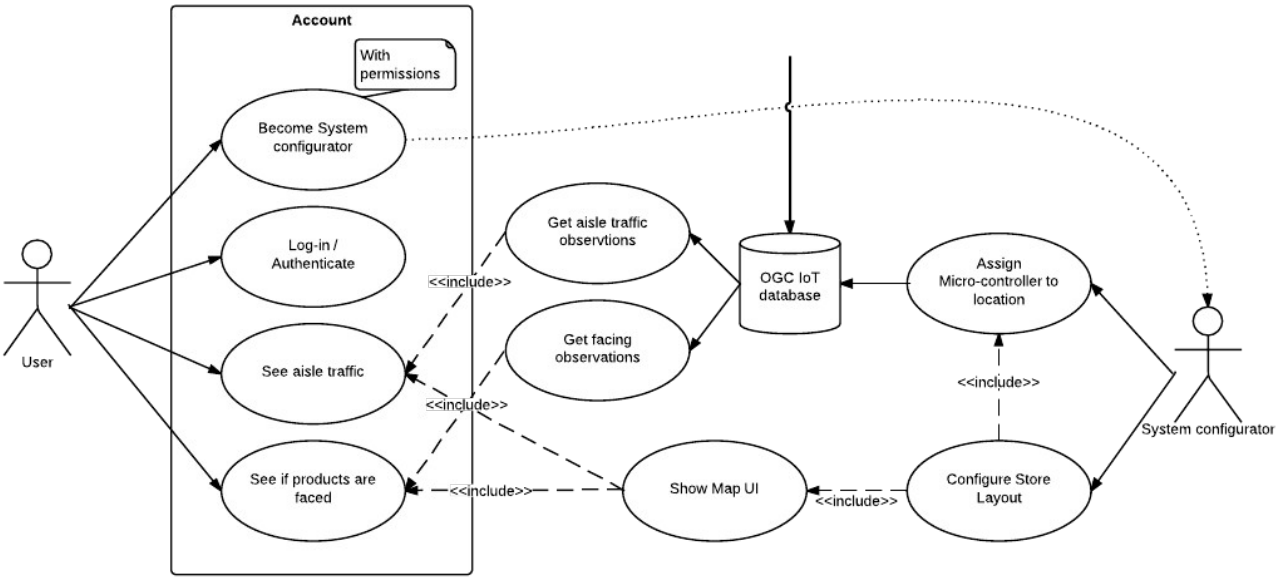

Figure 3: Use-case diagram for the website

In the diagram, the actors are represented by the “stick-man” figures, whereas the use cases are depicted by the ellipsoidal elements. The other elements, such as the database, exist to show the link between the shelf prototype and website use cases. The “Account” entity, and “With permissions” note are shown for clarity purposes.

Having completed the diagram, the team prepared a use cases for each of the identified actions that the system will be responsible for. See Tables 1 and 2 below:

Table 1: Shelf prototype use cases

1.1	Update database with formatted data
1.2	Format data according to OGC IoT standard
1.3	Uniquely identify sensors
1.4	Send data to micro-controller

Table 2: Website use cases

2.1	Assign location to a micro-controller
2.2	Configure a store layout
2.3	Get aisle traffic observations
2.4	Get facing observations
2.5	Get aisle traffic
2.6	See if products are faced
2.7	Show map UI
2.8	Log In/Authenticate
2.9	Become System Configurator

The majority of these use cases can be seen in detail in Appendix A and B. While the initial phase of use case analysis has been completed, the vision for the system is not set in stone. As noted previously, these are subject to change with the evolution of the project.

Having defined use cases opens up the possibility of beginning on UML case diagrams. This will be the next step in project planning to be tackled by the team. So far, only initial research into the requirements of making a UML diagram has been completed. For each use case, classes will be created to define the attributes and methods/operations needed to achieve the goals of the use case. The UML Class diagram will be created in UML 2.2 style as defined by the Object Modeling Group (OMG) [8] [9].

Location-Aware Shelf System (LASS) Prototype

In order to advance the hardware side of the design, an official equipment list has been prepared after much planning and budgeting of costs. Previously, the team had obtained sufficient equipment with which to experiment, including a micro-controller (Raspberry Pi) and two transistors. Although the team has conducted some testing with the devices available, our primary concern is to obtain a few PIR Motion Sensors and Magnetometers, with the help of Dr. Steve Liang, in order to start testing sensor capabilities and designing an appropriate electronic circuit. A Passive Infrared (PIR) motion sensor is an electronic sensor which measures infrared light radiating from objects in its field of view and it is used to sense movements of people, animals, or other objects. An example of a PIR Motion Sensor can be seen in Figure 4 below [10].

Figure 4: Example of Passive Infrared Motion Sensor

A magnetometer is a measuring instrument that is used to measure the strength and direction of magnetic fields. A magnetometer can be used to check the shelf facing as they can detect magnetic metals. For the purposes of our project, we aim to mount these magnetometers onto self facing shelf racks, which will reach specific magnetometer readings based on whether or not the shelf is empty. An example of a magnetometer can be seen in Figure 5 [11]:

Figure 5: Example of Magnetometer Sensor

A simple schematic of how these sensors are set up with respect to our shelf can be seen in Figure 6 below:

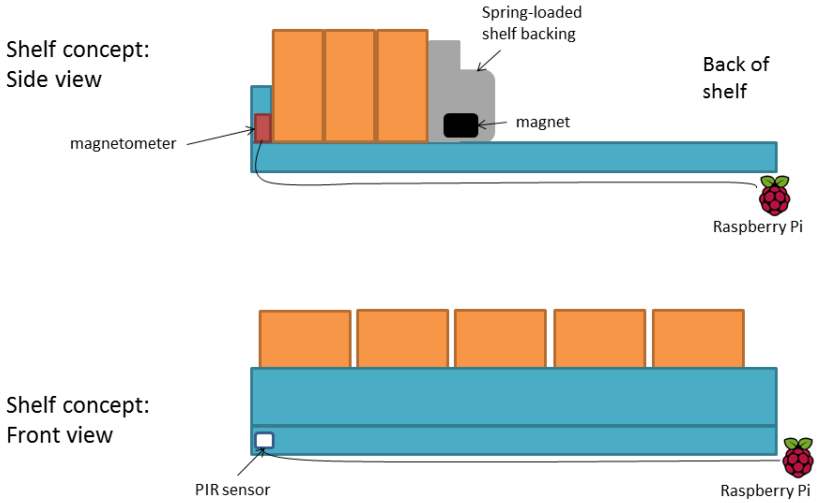

Figure 6: Schematic of shelf prototype system

In addition to hardware specifications, the team was also given access to Dr. Steve Liang's database on GitHub (OGC IoT GitHub) [3]. Time has been investing in learning and practicing with Python programming, and some skeleton code has been written, which will enable the creation of 'things' and update them to the database. From this we were able to observe the relationships between various entities belonging to each 'Thing' that has been created. This understanding is vital to our completion of the skeleton code and to the transfer of our collected data to the website.

Specifications

The following two lists of functional and non-functional specifications were compiled based on our previous literature review [5] for the Hardware section. These specifications seek to define the direction with which our final product will be developed.

Functional

1. The proposed shelf should be able to tell when a product is no longer faced. For example, in the case of spring load racks, magnets could be attached to the back of the spring loaded wall of the rack, and magnetometers at the face of the shelf could be fixed such that the magnetometers reach specific readings when the shelf is nearly empty.
2. The proposed shelf should be able to determine if somebody is standing in between the length-wise area of the shelves, such that it is possible to know whether a customer is standing between the two edges of the shelf-space. This makes it possible to know where in an aisle a customer might be. An example might be to use range or motion sensors at each edge of the shelf. If a customer crosses one side but not the other, this could indicate that they are in between both sensors.
3. The shelf should be sized similarly to existing store shelves, and should not provide any lower limits on the amount of stock that can be placed on a shelf. In this same way, the shelf should not obstruct or otherwise place limits on the amount of space

between aisles compared to that of existing shelves.

4. The final device should not break without internet access, but should expect reliable internet infrastructure in place in order to perform.

Non-Functional

1. The final product should conform to the Open Geospatial Consortium for IoT standard. Specifically, the data output from the hardware should be interoperable with a conformant database built from this standard.
2. Relatively stable internet access should be available (either through a wired or wireless connection) so that the shelf can update its status as frequently as possible.
3. Power to the shelf should be made available. For a single shelf consisting of one Raspberry Pi controller, a 700 mA (5V) power source is necessary to operate.

Web-Based Interface for LASS

The key word that many take away when imagining the Internet of Things is the *internet.* The internet has exploded in use and popularity in recent years, and plays an important part in information exchange. Therefore, as a corollary to our LASS prototype which takes measurements in from attached sensors, we are likewise developing a web-based user interface with which to interact with our shelf systems. This interface will exist as a website where targeted users can log in, view, and interact with the hardware through a web browser.

Specifications

The following two lists of functional and non-functional specifications were compiled in our previous literature review [5] for our website design. These specifications seek to define the direction with which our final product will be developed.

Functional

1. The ability to view shelf facings in real time, and determine where products in the store need to be restocked.
2. The ability to track customers entering and exiting each respective aisle, and determine how long each customer stayed in that section of the store.
3. The ability to generate some basic analytics based on data from the sensors. This can include items such as determining hotspots or busy times around the store, analyzing which parts of the store are more popular (most shelves empty), and other organizational information, such as busy times within the store, or average time left un-faced.

Non-Functional

1. The system should be painless and simple to introduce. Specifically, it should not require the user to do anything beyond setting up the hardware. For this reason, a central website that distributes management of multiple entities is preferred to a more decentralized system, where entities would set up their own specialized hardware for their application.
2. The system should be secure and prevent unauthorized entry. This is to protect consumers from privacy concerns, as well as to protect both managers and the data server from malicious entities.
3. The system should be robust enough that events such as power loss or switching

products between shelves should not produce any significant overhead to the system.

4. The system should be easy to manage and should provide managers with the ability to readily pull up any information from a given store or branch that they might choose.
5. The system should have received an extensive amount of testing, preferably upwards of 30 hours of actual use testing and bug-fixing. As the project expands the test requirements will also be expanded to maximize ease-of-use of the final product.
6. All of the software should operate using the OGC IoT SWG standard, and should be published as free, open-source software for anyone to access and improve upon.

Completed Work-to-Date

Aside from the above listed specifications, and the aforementioned use cases describing the interactions of the system as a whole, other work has been started in the interest of rapidly developing the web-based interface of our project. In particular, research regarding web technologies and frameworks has been progressing far ahead of schedule, thanks in part to splitting the team and using a divide-and-conquer approach for the remainder of the project milestones.

With regards to research, the website team has decided to develop the final system using the following popular, established web technologies:

- HTML + CSS + Javascript
- Bootstrap [12] for dynamic CSS
- jQuery [13] for dynamic animations and effects in our user interface (UI)
- D3JS [14] for visualization of our sensors

Of the above, the website team has already learned and become comfortable with HTML, CSS, and Javascript as development languages, and jQuery as a standard development library. Thus, much of the research for implementing the website is done, all that remains is to learn the remaining two technologies (Bootstrap and D3JS) and implement our use cases as defined in Appendix B.

Technical Demo of OGC Database Interaction

To demonstrate the progress made in understanding the OGC IoT SWG standard, and to exemplify how measurements are to be read off of the server from live data, a small technical demo page was created with a few interactive elements to show live data being read from the server in real-time. An example of this page can be seen in Figure 7 below:

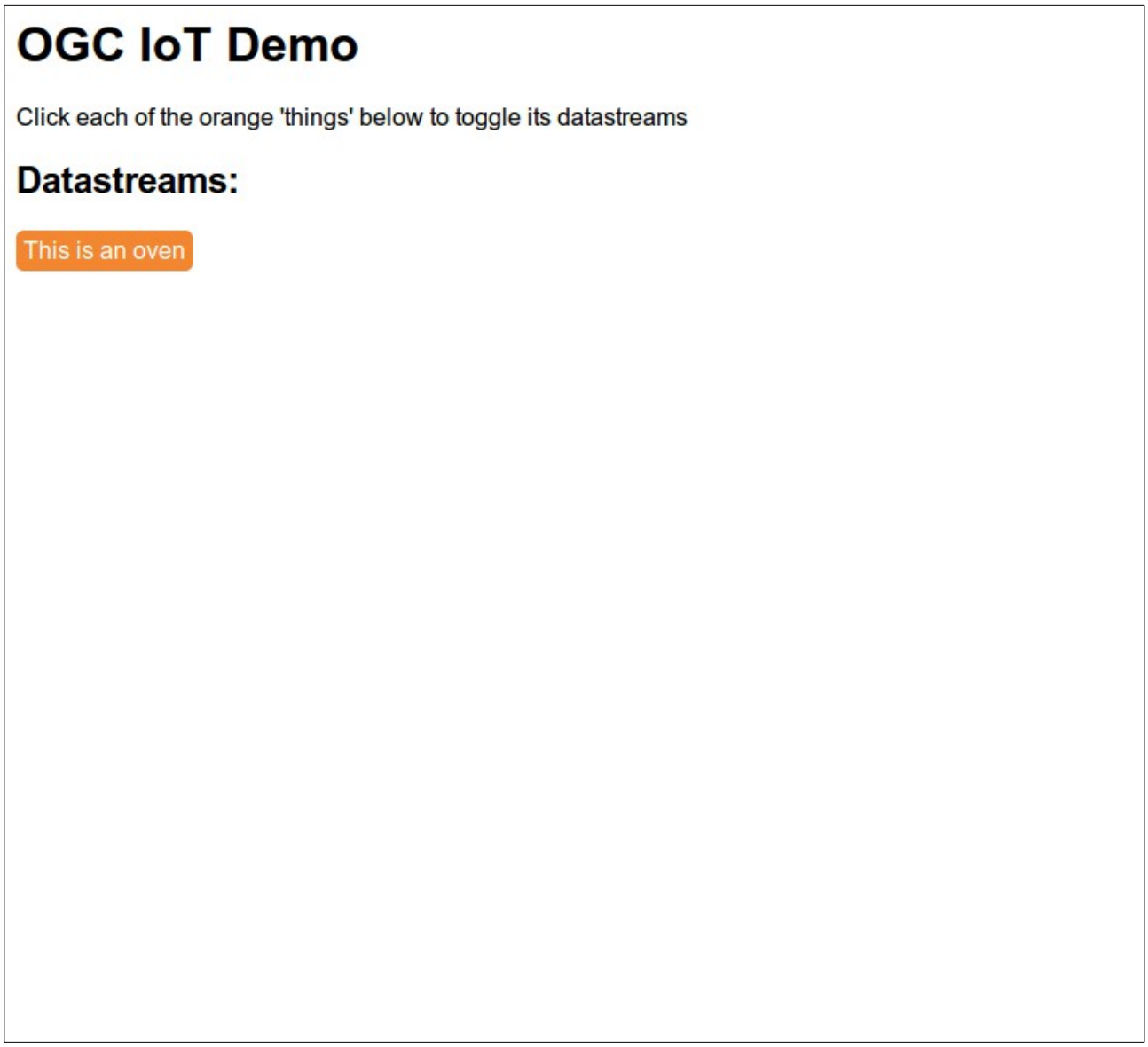

Figure 7:An example of the web page rendered before user interaction

From the above webpage, clicking on the orange button shown above yields the following data-streams pulled from the provided server, seen in Figure 8:

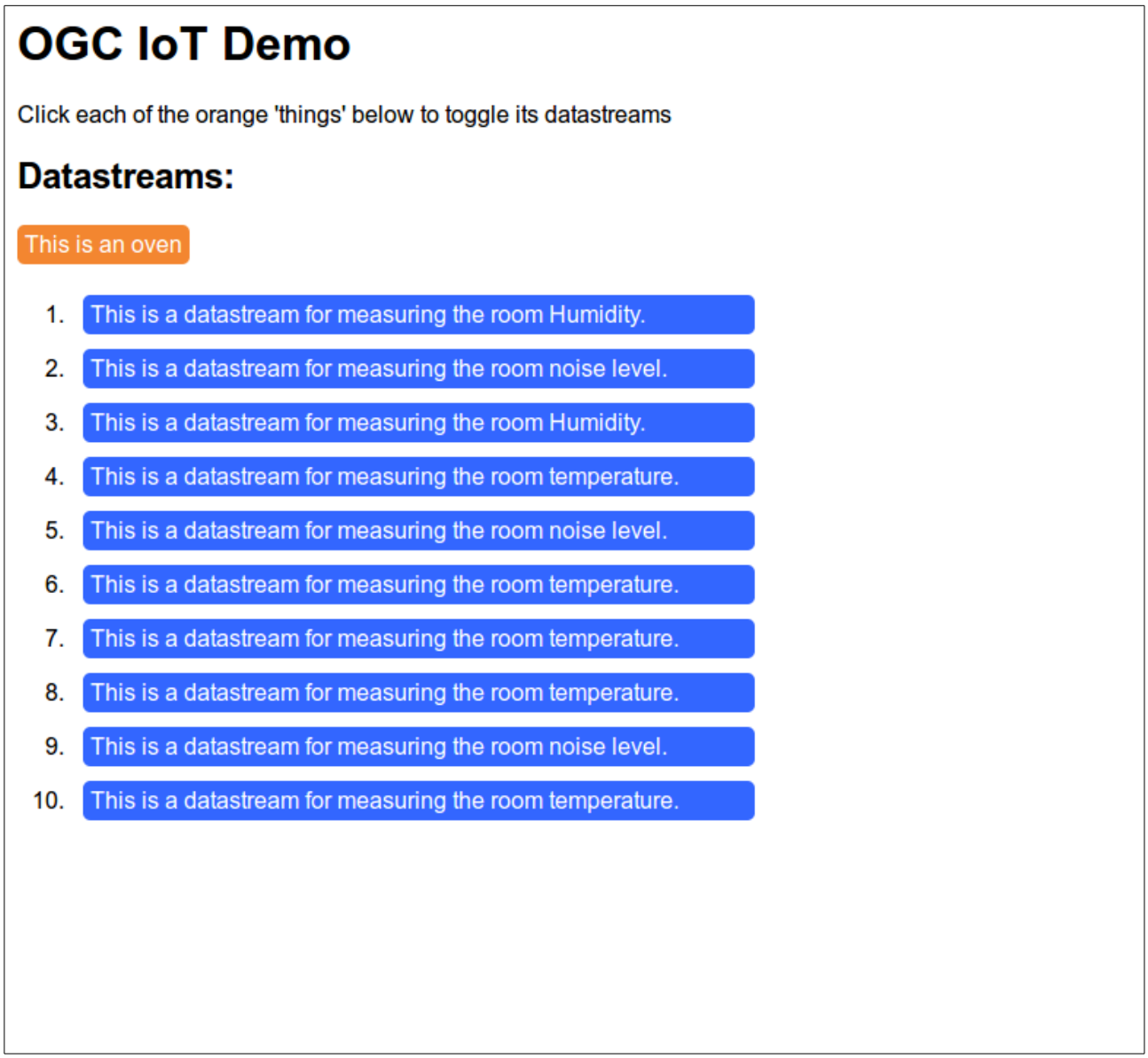

Figure 8: Dynamic loading of observation content (descriptions of data-streams shown)

While not directly applicable to the end goals of our interface, this example shows our progress in pulling data directly from the server, which is one of the larger hurdles in accomplishing our end goal. The full code can be found on the group Github repository [15], or can be seen at http://jsfiddle.net/XkBAm/1/.

Schedules, Risks and Status

The project has gone through a significant change from the original forecasted schedule of events. This was primarily due to a lack of experience in design, development procedures, and project management within the team. Although the tasks completed did not align well with the original timeline found in the project proposal, the team remains confident that the goals of the project will be met by the final deadline.

The greatest oversight in the predicted project velocity was in identifying and estimating the initial stages of the project. With a loosely defined project statement, the team spent a considerable amount of time deciding on an application for which an IoT solution was appropriate. The tasks

thats followed, such as deciding on which micro-controller to use, which sensors to use, what website framework to work with and learning the OGC's proposed IoT API were all time consuming tasks that had originally appeared to be small amounts of work in terms of technical deliverables. However, they did result in a greater knowledge and understanding of each of the components that make up the project, as outlined in the preceding section. These were not all anticipated steps, and thus, were not shown in the original Gantt Chart.

Although the team had planned on doing the project in two phases, sensor prototype development followed by website development, deviation from the original plan occurred. At week 8, the team decided to split into a team of three working on hardware implementation, while a team of two began work on the product's website. This made a considerable change to the timeline of the project.

Examining the Gantt Chart from the project proposal shows that the following milestones in Table 3 below were projected to be done by December 6th:

Table 3: Milestones to be completed before end of term (December 6th)

Milestone	Status	Comments
Determine location based application	Complete	The application chosen was Location-Aware Shelf System (LASS)
Design sensor setup - Determine technical requirements - Acquire Sensors - Create prototype - Test prototype	 Complete In Progress Not started Not started	The technical requirements were determined as part of the deliverables for the literature review. The required sensors have been sent for approval but not acquired.
API-Sensor interface - Research into languages/specifications - Collect test set of data	 Complete Not Started	While the language and specifications have been researched, having not purchased sensors leaves this task unfinished.
Project Proposal	Complete	
Literature Review	Complete	
Technical Deliverables - Written Report - Oral presentation	 Complete Complete	

The following milestones in Table 4 were projected to have been started by December 6th:

Table 4: Milestones to have been started by end of term (December 6th)

Milestone	Status	Comments
Create Library to interface with database	In Progress	Using simulated data, information has successfully been PUT to the database

Table 5 below depicts milestones that were projected to begin *after* December 6th, but have been started due to the introduction of a split team focus:

Table 5: Milestones projected to being at the start of the second term, but have already been started

Milestone	Status	Comments
UI Development - Research required frameworks - Create prototype of UI - Attach to database - Test UI - Polish / Revise UI - Create video of UI in use	 Complete Not Started In Progress Not Started Not Started Not Started	Progress to date focused on learning website development, and interfacing with the database. A mock website was created, but UI development has not yet begun.

Lastly, we can see Table below, which describes milestones and sub-tasks that were not initially predicted, but were completed before December 6th:

Table 6: Milestones not initially accounted for but have been completed or are in progress before the end of term (December 6th)

Milestone	Status	Comments
Define Use Cases - Hardware Use Case Diagram - Hardware Use Cases - Website Use Case Diagram - Website Use Cases	 In Progress In Progress In Progress In Progress	While the Use Cases attached in Appendix A and B at the end of the report are mostly complete, they have yet to be finalized by our supervisor, and some additional ones remain to be added at this moment in time.
Learn web development fundamentals - HTML - CSS - jQuery	 Complete Complete Complete	This milestone was for the two team members focused on web development.
Learn introductory Python programming	Complete	This milestone was for the three team members focused on hardware development
Explore and understand database structure and commands	Complete	
Create demonstrative proof of concept - Post data to server - Display data on website	 Complete Complete	This milestone was done for the oral presentation

Due to the large change in the project direction, a new Gantt chart has been created. The time that has passed now reflects the project milestones that have been accomplished thus far, and a re-evaluated version of the future milestones can now be seen past week 12. This Gantt chart can be seen in Figure 9, on the following page:

Figure 9: Updated Gantt chart of project timeline. Current time is marked by the solid gray bar, tasks that have been set back are marked in maroon.

Looking at the original Gantt chart seen in the proposal [16], this one has several differences, most notably that it is far less vague regarding project requirements and scope, which have been more properly been defined since that time. The major differences are:

- The original *Application Design* section was expanded to the new *Project Conceptualization & Planning* section, which lays out several sub-tasks within the milestone, including:
    - Research into the value of chosen application
    - Research applicable sensors
    - Produce specification sheet(s)
    - Develop and flesh-out our class model diagrams.
    - Build use cases to define the structure and interactions between our software elements
- The *Design sensor setup* sub-task has been split from the planning phase of the project, and more appropriately placed into the *Application Prototype Development* section. The sub-tasks of this milestone have likewise been merged with the *API-sensor interface* milestone, since a large portion of the work done in creating our prototype will revolve around ensuring that our prototype can properly communicate using with the provided OGC IoT standard.
- The website development has been consolidated, and proper testing steps (including a usability survey) have been added as sub-tasks.

Potential Risks

Some of the potential challenges we risk running into can be summarized in the following list:

1. Given that we have delayed ordering sensors until really late, we potentially run the risk of losing much of our time waiting for the sensors to ship. Our application prototype currently allots for some time to be lost (see Figure 9), however, there is only approximately a three week window where this is acceptable. Should the sensors be delayed by more than three weeks (from their regular shipping time, 4-6 weeks), there may be serious consequences to the final leg of our development. To mitigate this, we are using the inputs and outputs specified by our use-cases to simulate data read in from the sensors which we currently do not have. In this way, we should still be able to make a functioning system, which should allow other pieces of the system to interact even if the physical sensors themselves are missing.
2. Since our group has split into two teams working simultaneously, one focused on developing the prototype, and the other focused on developing the web-based interface, we run the risk of developing orthogonal to each other's work. In this case, we require strict definition and adherence to our use cases in order to avoid running into a situation where the input of one of our components is not coherent with the remainder of our system. In this way, by mitigating this risk, we also save ourselves from code duplication, and prevent ourselves from running in circles re-writing modules and objects.
3. One significant challenge during this term was gaining some momentum within the project, specifically because much of the legwork this term favoured planning, careful research, and conceptualization over hard deliverables. This resulted in appearing to have very little done on the surface, while having established a

reasonable framework through our use-cases and documentation with which to develop our project further. Given that there are a larger number of breaks in the winter term (Christmas break, Reading week, Easter, etc...), we run the chance of losing our momentum and potentially falling behind. What's more, with this being the last term before (hopefully) graduation for the group, there is a chance that work from prioritizing other classes and electives constitutes a significant barrier to getting work done fluidly and efficiently. For this reason, a more stringent time input will likely be enforced in the new term, and expectations for weekly output will attempt to be measured to some standard. Moreover, much of the work into the finalization and development of our use case modules will attempt to be done over these breaks, to try and mitigate the otherwise demanding nature of engineering.

4. One risk that was not fully explored in this document or in others we have published regarding cost. It is assumed until this point that our prototype will be built, but little thought has been given to cost aside from ordering the sensors. This is a significant risk in that if we want our final product to be successful and have impact, we need to consider the cost of the final device and the associated ease-of-use and value it provides. The value itself has been explored within our previous Literature Review [5], but has not necessarily been weighed against the cost of individual sensors (or that of mass-produced products, which would likely bring about better results). Part of the final report should perhaps explore what was learned with regards to the number of sensors required for effective testing, and what the lower limits are in terms of cost reduction for future continued development and implementation.

Conclusion

Overall, the evolution of the project, as well as the management principles of the project have shifted greatly from the original intent planned back during the project proposal. Accomplished milestones within our planning phase, prototype development, and web-based interface (website) development were explored, and progress was evaluated. Some specific roadblocks were encountered that hindered our expected progress in the planning and prototype development milestones; specifically, use-case diagrams became necessary to mitigate further risk, and a late determination and order time of sensors has the potential to set back the progress of the prototype development. However, despite these challenges, by reorganizing the group into two teams to tackle the remaining tasks, significant headway was gained in the website development phase, particularly with regards to the research required to learn web technologies and frameworks.

Despite the progress made, some things could have been handled differently, most notably the amount of contact kept with our supervisor (Dr. Steve Liang) throughout the semester. It was difficult attempting to get a hold of Dr. Liang, in particular due to his travel and work schedule. We were often limited to bi-weekly meetings, which in the future may be more beneficial as weekly meetings. The amount of software training also proved challenging, as many of the group members were not acclimated to Github so easily. In retrospect, more time could have been spent properly training each group member to ensure that each member understood how to use and manipulate files and revisions through using git and Github.

Appendix A: Shelf Prototype Use Cases

Use Case ID:	1.1
Use Case Name:	Update database with formatted data
Created by:	AC
Date Created:	24-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	Raspberry Pi and sensors as they store and collect data
Description:	Once the hardware is set up and working, data will be collected by the actors and stored. As this happens it should be formatted if need be (use case 1.2). The database should then be accessed and updated with new, formatted data. This will eventually be implemented in real time.
Preconditions:	1. Sensor is connected and collecting data
	2. Database is accessible
	3. Data has been successfully formatted
Postconditions:	1. Database received the new values successfully
Priority:	High during real time updates, otherwise normal
Frequency of Use:	High
Normal Course of Events:	1. Data is collected and stored by actors
	2. Data is rendered into a database compatible form
	3. Database is accessed and data is transferred
Alternative Courses:	1.a Data is collected but not stored
	1.a.1 User is notified of the error
	3.a Data is transferred but is not formatted correctly
	3.a.1 User is notified
Exceptions:	
Includes:	1.2 Format Data
Special Requirements:	
Assumptions:	1. Assumes that the collected data needs to be formatted. Will be verified through testing.
	2. Assumes that the database (can be accessed and) does not clear itself every 10 minutes.
Notes and Issues:	Much of this may change as the group begins working with sensors.

Use Case ID:	1.2
Use Case Name:	Format data according to OGC IoT standard
Created by:	AC
Date Created:	25-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	Raspberry Pi and sensors as they store and collect data
Description:	Data collected by sensors will be formatted so that it can be easily updated to the database with use case 1.1
Preconditions:	1. Sensor is connected
	2. New data is being gathered and stored
Postconditions:	1. Database received the new values successfully
Priority:	High during real time updates, otherwise normal
Frequency of Use:	High
Normal Course of Events:	1. Data is collected and stored by actors
	2. Data is rendered into a database compatible form
Alternative Courses:	1.a Data is collected but not stored
	1.a.1 User is notified of the error
Exceptions:	
Includes:	1.1 Update database
Special Requirements:	
Assumptions:	
Notes and Issues:	Much of this may change as the group continues working with sensors.

Use Case ID:	1.3
Use Case Name:	Identify sensors
Created by:	AC
Date Created:	25-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	Raspberry Pi and sensors attached to it
Description:	All the sensors collecting data need to be recognized and given a unique ID so that their data may be distinguished even after it is formatted and updated to the database
Preconditions:	1. Sensor is connected and recognized
Postconditions:	1. The data that was collected can be related to a unique ID that represents each sensor
Priority:	High during real time updates, otherwise normal
Frequency of Use:	High
Normal Course of Events:	1. Sensors are connected and recognized
	2. Sensors are assigned a unique ID
	3. Data is collected and the ID of the sensors corresponds to it
	4. All this is relayed to the formatting use case (1.2)
Alternative Courses:	1.a Data is collected but not stored
	1.a.1 User is notified of the error
Exceptions:	
Includes:	1.2 Formatting Data
Special Requirements:	
Assumptions:	
Notes and Issues:	Much of this may change as the group continues working with sensors.

Use Case ID:	1.4
Use Case Name:	Send data to micro-controller
Created by:	BT
Date Created:	5-Dec-13
Date Last Updated:	
Last Updated by:	
Actor:	Sensor
Description:	When observations for a sensor are made, the sensor will begin to register a new value in it's datastream. This change will need to be acknowlged by the microcontroller and read into the program used to format and send observations to the database.
Preconditions:	1. Sensor is connected to micro-controller
	2. Sensor is uniquely identified, and datastream has been created
	3. Observations are being collected
Postconditions:	1. Program running on micro-controller receives new observation
Priority:	High
Frequency of Use:	High
Normal Course of Events:	1. Observation is collected by sensor
	2. Micro-controller reads new observation
Alternative Courses:	
Exceptions:	2.ex Micro-controller does not read new observation
	2.ex.1 Micro-controller reads next observation, ignoring the lost observation
Includes:	1.1 Uniquely identify sensor
Special Requirements:	
Assumptions:	
Notes and Issues:	Much of this may change as the group continues working with sensors.

Appendix B: Website Use Cases

Use Case ID:	2.1
Use Case Name:	Assign a location to a microcontroller
Created by:	BT
Date Created:	24-Nov-13
Date Last Updated:	30-Nov-13
Last Updated by:	BT
Actor:	Person setting up or changing sensor configuration
Description:	The microcontroller will have no way of knowing it's geographic location, so this should be configurable by a person. Once connecting a microcontroller, the user should be able to see that a new 'thing' has been added to the sensing network (Do they register themselves or do they need to be registered?). This will have a unique ID and can be assigned a location based on a map of the store.
Preconditions:	1. User is logged in/authenticated
	2. System describing spatial location of shelves exists (may include map)
	3. Microcontroller has registered as a 'thing' in the database
Postconditions:	1. Entity type: Location is set
	2. New/no-location 'thing' flag removed
Priority:	Low
Frequency of Use:	High during initial set-up, low afterwards
Normal Course of Events:	1. User is made aware of 'thing' that has no location
	2. User enters a mode where the location of the 'thing' can be added (either via map or manual entry)
	3. User is shown location on map to verify that location is correct
	4. Location is saved to database
	5. Sensor's location can be accessed/displayed
Alternative Courses:	1.a 'Thing' has a location, but is being moved to a new location
	1.a.1 Remove Location attribute of associated sensors
	1.a.2 Remove location of microcontroller
Exceptions:	2.ex User enters a location that is not valid
	2.ex.1 User is notified of the error and asked for a new location
	4.ex Database is not available
	4.ex.1 User is notified of the error
Includes:	2.8 Log in/authenticate
	2.2 Configure a store layout
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	2.2

Use Case Name:	Configure a store layout
Created by:	BT
Date Created:	24-Nov-13
Date Last Updated:	30-Nov-13
Last Updated by:	BT
Actor:	High during initial set-up, medium during operation
Description:	There needs to be a way for a user to construct a store layout system in order to locate sensors. This will be based on a typical grocery store shelf. Attributes can be added to increase the specificity of the model, but a bare-bones requirement will be enforced to ensure data can be mapped to the right area. This system will work with a visual representation (map) to provide context and help users.
Preconditions:	1. User is logged in/authenticated
Postconditions:	1. Store layout is saved
	2. Spatial reference system can be used for other use-cases
Priority:	Medium
Frequency of Use:	Medium
Normal Course of Events:	1. User enters store layout creator
	2. User changes aisle configuration
	3. User changes facing configuration
	4. Spatial reference system is updated
Alternative Courses:	
Exceptions:	
Includes:	2.8 Log in/authenticate
Special Requirements:	
Assumptions:	Some sort of quad-tree data structure can accommodate the addition/removal/editing of shelve layout
Notes and Issues:	Functionality of this feature may need to be limited due to complexity, experience and time-restraints. In the worst case scenario, the website only features one spatial reference system that cannot be changed.

Use Case ID:	2.3
Use Case Name:	Get aisle traffic observations
Created by:	BT
Date Created:	24-Nov-13
Date Last Updated:	Novemeber 30,2013
Last Updated by:	BT
Actor:	Internal, part of 2.5 See aisle traffic
Description:	Observation data from the motion sensors needs to be accessible so that it can be displayed on the map. This data is located in the database, where each sensor will have a unique ID, associations with other objects, and other attributes. Using these associations to filter, sensor data needs to be requested with jquery.
Preconditions:	1. User is logged in/authenticated.
	2. A 'thing' with motion sensors exists in the database
	3. A datastream belongs to said 'thing', made from observations belonging to said sensor
Postconditions:	1. Data from motion sensor is retrieved from the database
Priority:	High
Frequency of Use:	High
Normal Course of Events:	1. Webpage gets new data from database using jquery
	2. Data is assigned to variable which can be used to display data
Alternative Courses:	1.a No new data exists
	1.a.1 No new data is assigned to variable
Exceptions:	
Includes:	
Special Requirements:	Each motion sensor observation is associated with a micro-controller from which it originated. This will be used to determine the location to show the data on the map.
Assumptions:	
Notes and Issues:	

Use Case ID:	2.4
Use Case Name:	Get facing observations
Created by:	BT
Date Created:	30-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	Internal, part of 2.6 See if products are faced
Description:	Observation data from the magnetometer/(other sensor?) needs to be accessible so that it can be displayed on the map. This data is located in the database, where each sensor will have a unique ID, associations with other objects, and other attributes. Using these associations to filter, sensor data needs to be requested with jquery.
Preconditions:	1. User is logged in/authenticated.
	2. A 'thing' with a sensor used for facing detection exists in the database
	3. A datastream belongs to said 'thing', made from observations belonging to said sensor
Postconditions:	1. Data from sensor used for facing detection is retrieved from the database
Priority:	High
Frequency of Use:	High
Normal Course of Events:	1. Webpage gets new data from database using jquery
	2. Data is assigned to variable which can be used to display data
Alternative Courses:	1.a No new data exists
	1.a.1 No new data is assigned to variable
Exceptions:	
Includes:	
Special Requirements:	Each facing detection sensor observation is associated with a micro-controller from which it originated. This will be used to determine the location to show the data on the map.
Assumptions:	
Notes and Issues:	

Use Case ID:	2.5
Use Case Name:	See aisle traffic
Created by:	BT
Date Created:	30-Nov-12
Date Last Updated:	
Last Updated by:	
Actor:	Manager, Store Clerk
Description:	A user (Manager or store clerk) should be able to see if the aisle traffic on a map of the store. This should update automatically without input from the user, so they can have an idle screen showing the data as it become available. If they wish, this data can be filtered in some useful ways. It will have both a graphical and numerical version of the data, displayed in an attractive and easy to understand format.
Preconditions:	1. User is logged in/authenticated.
	2. A 'thing' with motion sensors exists in the database
	3. A datastream belongs to said 'thing', made from observations belonging to said sensor
	4. Aisle traffic observations are being retrieved by the database and assigned to a variable
Postconditions:	1. Traffic information is shown on the map in a graphical form
	2. Traffic information is shown on the map in numerical form
Priority:	Medium
Frequency of Use:	High
Normal Course of Events:	1. User navigates to map view
	2. Observations are shown without input from user
Alternative Courses:	2.a User adds filter input
	2.a.1 Data displayed based on the filters that the user has configured
	2.b There are no observations
	2.b.1 There is an indication that the sensor is working, but that it is in an idle state
Exceptions:	2.ex Observations are not being shown on map
	2.ex.1 User is shown an error message of some kind, most likely propagated from use case 2.4 get aisle traffic observations
Includes:	2.8 Log in/authenticate
	2.1 Assign a location to a micro-controller
	2.2 Configure a store layout
	2.3 Get aisle traffic observations
	2.7 Show map UI
Special Requirements:	
Assumptions:	

Use Case ID:	2.6

Use Case Name:	See if products are faced
Created by:	BT
Date Created:	30-Nov-12
Date Last Updated:	
Last Updated by:	
Actor:	Manager, Store Clerk
Description:	A user (Manager or store clerk) should be able to see if the products are faced in an area of the store via the map UI. This should update automatically without input from the user, so they can have an idle screen showing the data as it become available. If they wish, this data can be filtered in some useful ways. It will have both a graphical representation of the data, displayed in an attractive and easy to understand format.
Preconditions:	1. User is logged in/authenticated.
	2. A 'thing' with a sensor used for facing detection exists in the database
	3. A datastream belongs to said 'thing', made from observations belonging to said sensor
	4. Facing information observations are being retrieved by the database and assigned to a variable
	5. Store layout has been configured
Postconditions:	1. Facing information is shown on the map in a graphical form
Priority:	Medium
Frequency of Use:	High
Normal Course of Events:	1. User navigates to map view
	2. Observations are shown without input from user
Alternative Courses:	2.a User adds filter input
	2.a.1 Data displayed based on the filters that the user has configured
Exceptions:	2.ex Observations are not being shown on map
	2.ex.1 User is shown an error message of some kind, most likely propagated from use case 2.5 Get facing observations
Includes:	2.8 Log in/authenticate
	2.1 Assign a location to a micro-controller
	2.2 Configure a store layout
	2.4 Get facing observations
	2.7 Show map UI
Special Requirements:	
Assumptions:	
Notes and Issues:	

Use Case ID:	2.7
Use Case Name:	Show Map UI
Created by:	BT
Date Created:	30-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	Internal / User (manager, store clerk)
Description:	Once a store layout is configured, there should be a map UI where the user can see observation data in it's correct location. This map will include controls to filter observations in certain ways. The map should update as new observations become available.
Preconditions:	1. User is logged in/authenticated
	2. Store layout has been configured
Postconditions:	1. Map is shown
Priority:	Medium
Frequency of Use:	High
Normal Course of Events:	1. User navigates to map view
	2. Map is shown
Alternative Courses:	
Exceptions:	2.ex Map cannot be rendered from store layout
	2.1.ex Error message is shown
Includes:	2.8 Log in/authenticate
	2.2 Configure a store layout
Special Requirements:	Map enables 2.5 and 2.6 to be displayed. The map will need to update automatically
Assumptions:	
Notes and Issues:	

Use Case ID:	2.8
Use Case Name:	Log In/authenticate
Created by:	BT
Date Created:	30-Nov-13
Date Last Updated:	
Last Updated by:	
Actor:	User (Store clerk, Manager
Description:	In order to protect a group's data, authentication will be required to access an account. Depending on if the user is a store clerk or Manager, they can have different permissions.
Preconditions:	1. Account exists with username and password
Postconditions:	1. User is logged in and presented with their group's data
Priority:	Low
Frequency of Use:	High
Normal Course of Events:	1. User enters username and password
	2. Username password combo is verified and user is logged in
Alternative Courses:	
Exceptions:	2.ex Username password combo is incorrect
	2.ex.1 User is not logged in, and informed that they entered the wrong User/pass
Includes:	
Special Requirements:	
Assumptions:	
Notes and Issues:	There will be a different level of permission for a manager and store clerk

Use Case ID:	2.9
Use Case Name:	Become System configurator
Created by:	BT
Date Created:	30 November, 2013
Date Last Updated:	
Last Updated by:	
Actor:	Manager
Description:	If a manager logs in, they should have the powers of a system configurator. This will allow them to change store/micro-controller layout properties.
Preconditions:	1. Manager is logged in
	2. Permissions are set
Postconditions:	1. Manager has access to store layout
	2. Manager can assign a location to a micro-controller
Priority:	Medium
Frequency of Use:	Low
Normal Course of Events:	1. Manager logs in
	2. Manager navigates to a configuration options section/page
	3. Store layout and micro-controller location are edit-able
Alternative Courses:	
Exceptions:	
Includes:	2.8 Log in/autenticate
	2.2 Configure a store layout
	2.1 Assign a location to a microcontroller
Special Requirements:	
Assumptions:	
Notes and Issues:	

References

[1] Margaret Rouse, “What is the Internet of Things (IoT)?,” *whatis.com*, Jul-2013. [Online]. Available: http://whatis.techtarget.com/definition/Internet-of-Things. [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] “OGC SensorThings API,” *OGC SensorThings API*, 2013. [Online]. Available: http://ogc-iot.github.io/ogc-iot-api/. [Accessed: 05-Dec-2013].

[4] “OGC Standards and Supporting Documents,” *OGC Standards and Supporting Documents*. [Online]. Available: http://www.opengeospatial.org/standards. [Accessed: 06-Dec-2013].

[5] 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.

[6] Mirela Popa, Leon Rothkrantz, Zhenke Yang, Pascal Wiggers, Ralph Braspenning, and Caifeng Shan, “Analysis of Shopping Behavior based on Surveillance System,” in *Analysis of Shopping Behavior based on Surveillance System*, 2010, pp. 2512–2519.

[7] Pierre Chandon, J. Wesley Hutchinson, Eric T. Bradlow, and Scott H. Young, “Does In-Store Marketing Work? Effects of the Number and Position of Shelf Facings on Brand Attention and Evaluation at the Point of Purchase,” *J. Mark.*, vol. 73, pp. 1–17, 2009.

[8] Object Modeling Group, “OMG Unified Modeling Language InfraStructure,” *UML 2.2*, Feb-2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/. [Accessed: 05-Dec-2013].

[9] Object Modeling Group, “OMG Unified Modeling Language SuperStructure,” *UML 2.2*, Feb-2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/. [Accessed: 05-Dec-2013].

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

[11] Alex Louden, “Mechatronics Project - Sensors,” *Alex Louden*, 09-Aug-2010. [Online]. Available: http://alexlouden.com/2010/mechatronics-project-sensors/. [Accessed: 05-Dec-2013].

[12] “Bootstrap,” *GetBootstrap*. [Online]. Available: http://getbootstrap.com/2.3.2/. [Accessed: 05-Dec-2013].

[13] “jQuery - Write less, do more.,” *jQuery*. [Online]. Available: jquery.com. [Accessed: 05-Dec-2013].

[14] “D3: Data-Driven Documents,” *D3.js - Data-Driven Documents*. [Online]. Available: http://d3js.org/. [Accessed: 05-Dec-2013].

[15] Jeremy Steward, “ThatGeoGuy/ENGO500,” *ThatGeoGuy/ENGO500*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500. [Accessed: 05-Dec-2013].

[16] 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.

UNIVERSITY OF CALGARY

ENGO 500: GIS and Land Tenure #2

Progress Report

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

DEPARTMENT OF GEOMATICS ENGINEERING

SCHULICH SCHOOL OF ENGINEERING

02/14/2014

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

February 14, 2014

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

Dear Dr. Steve Liang:

Attached you will find a Progress Report detailing the efforts of the GIS & Land Tenure 2 group to fulfill the ENGO 500 course requirements. The report details the team's work towards the goal of creating an Internet of Things (IoT) application that interfaces with the Open Geospatial Consortium IoT specification, as well as provides an evaluation of the team's progress, and estimates remaining work.

Within the first month and a half of the semester, the team has managed to post observations from a sensor-microcontroller combination to the OGC SensorThings API, create the framework for a web-server, build an interactive webpage for users to configure the solution in physical space, and finally receive and display observations from the server.

Some of the tasks that were intended to be finished at this time have crept passed their projected deadline, and evaluation of the reasons behind this are discussed within the report. Overall, the team has progressed well and a clear path to completion has been identified.

Thank you, Dr. Liang, for taking the time to oversee our project. Your efforts to uphold weekly meetings with the team have greatly increased our productivity. If you have any questions about the contents of the report, or concerns of any other nature, please feel free to contact one of the team members at your convenience.

Sincerely,

Group GIS & Land Tenure 2

Table of Contents

Index of Figures

Index of Tables

Introduction

Purpose & Scope

The purpose of this document is to summarize and describe the progress of group “GIS & Land Tenure #2” to date. This report describes in detail the work done by the group with regards to the planning process of the project, the development of a prototype smart-shelf as part of the project design, and the development of a user-friendly web-based interface with which to interact with our shelf prototype [1].

Upon completion of describing the work done thus far in the project lifetime, the project schedule and timeline is evaluated. Milestones accomplished and problems encountered are discussed and evaluated, and finally a risk-assessment for the future of the project lifespan is considered. A final summary of the project status is stated, followed by a brief conclusion on the work done and goals to strive for in the remaining life of the project [1].

Project Overview

As mentioned in previous reports, the main focus of this project is about the concept of “Internet of Things”. The “Internet of Things (IoT)” is a scenario in which objects, animals or people are provided with unique identifiers and the ability to automatically transfer data over a network without requiring human-to-human or human-to-computer interaction [2].

This project, in particular, will be focusing on developing a Location Aware Smart Shelf (LASS), that can track customer interests or purchases throughout a grocery store. This will be developed as a Free, Open Source Software (FOSS) for developers who are looking to develop an “Internet of Things” using the Open Geospatial Consortium (OGC) Standard. The OGC Standards are technical documents that detail interfaces or encodings. Software developers often use these documents in order to build open interfaces and encodings into their product [3]. Thus, the project comprises two major components:

1. The Sensor Prototype component: comprises the development of a prototype smart shelf that can send sensor readings to the OGC IoT SensorThings data service.
2. Software / Web Component: Allows a user to interact / acquire readings from one or more of these prototype shelves that are mentioned above.

Location Aware Shelf System (LASS) Prototype

This section describes all of the progress made on the LASS prototype since mid-December. It begins with an overview of the major electronic components we have acquired and have been using so far. Following that, a description of the OGC SensorThings API Data Model for IoT applications is given, specifically highlighting the relevant components for our application. The next sub-section breaks down the different use cases that have been identified and how they have been implemented so far.

Status of Hardware Inventory

At this point, the hardware necessary for preliminary testing has been acquired. All of the programming and electronic assembly has been primarily accomplished with the following equipment:

- 1 Raspberry Pi and peripherals (monitor, keyboard, mouse)

- 1 breadboard and Pi cobbler
- 1 PIR motion sensor designed by seeed

The equipment being used is shown in Figure 1:

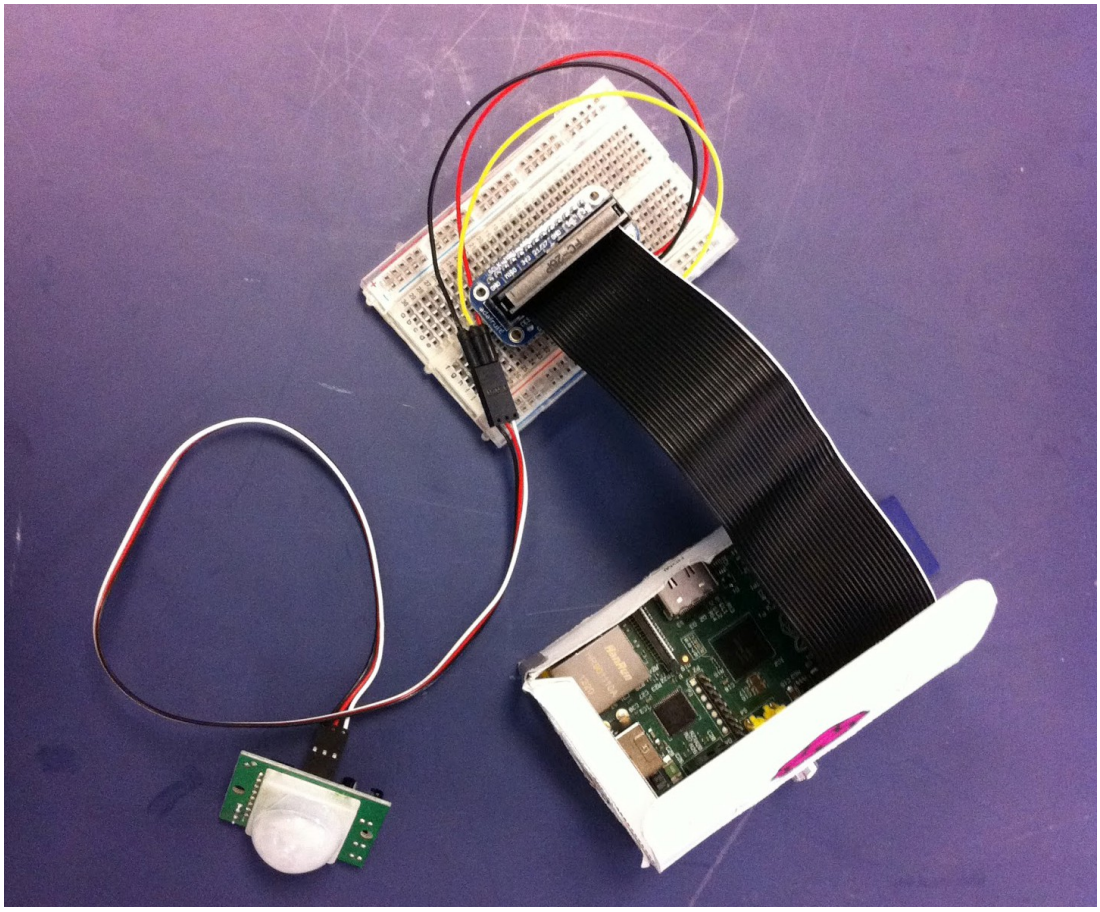

Figure 1: Current hardware being used, including a pseudo-infrared motion sensor, breadboard, pi-cobbler accessory and breakout cable, and Raspberry Pi

While the equipment lacks the physical shelf component, it has enough functionality to facilitate implementation of the key use cases. The current electronics setup was established based on following a tutorial found in the Adafruit Learning System [4]. This was used mainly to determine which GPIO pins should be used for the different PIR sensor wires.

Finally, additional sensors have been ordered, including more PIR motion sensors as well as photo interrupters, which will be used for detecting shelf stock. In this way, we have replaced our previous order list of magnetometers (which can measure any magnetic field) to photo interrupters, which will only pick up the electromagnetic signal of nearby elements through induction.

OGC SensorThings API – Data Model

One of the major requirements for this project is compliance with the OGC SensorThings API draft standard. As such, it is necessary for the prototype to fit into the defined IoT data model [5]. The entire IoT data model acts as a general umbrella which can encompass a large range of different IoT applications. Aside from the two core elements (a Thing and a Location), there are two main parts: the Tasking Profile and the Sensing Profile. Because the purpose of the Tasking Profile is to outline the data structure of a “Thing” which can be controlled by an application, this part of the data model

is not relevant to the LASS prototype. On the other hand, the Sensing Profile defines the components of an IoT application (both in terms of physical device and attributes) which can be acted on by four main functions – Create, Read, Update and Delete.

The general Sensing Profile, which is shown in Figure 2, has several entities. The entities which have been identified as primarily relevant and utilized in this stage of our project are highlighted in blue. In the context of our application, the Thing is the Raspberry Pi, the Datastream identifies/groups the motion sensor observations together and each Observation corresponds to a reading from the sensor. At this point, since only one sensor is being used, the Sensor entity has not yet been included. There is also no Observed Property entity either, because a description of the observed property is included in the Datastream and the units are not crucial in this application. Neither of the GeoJSON_Geometry objects (Location and Feature of Interest) were included either, since the location identification will be achieved via the web interface.

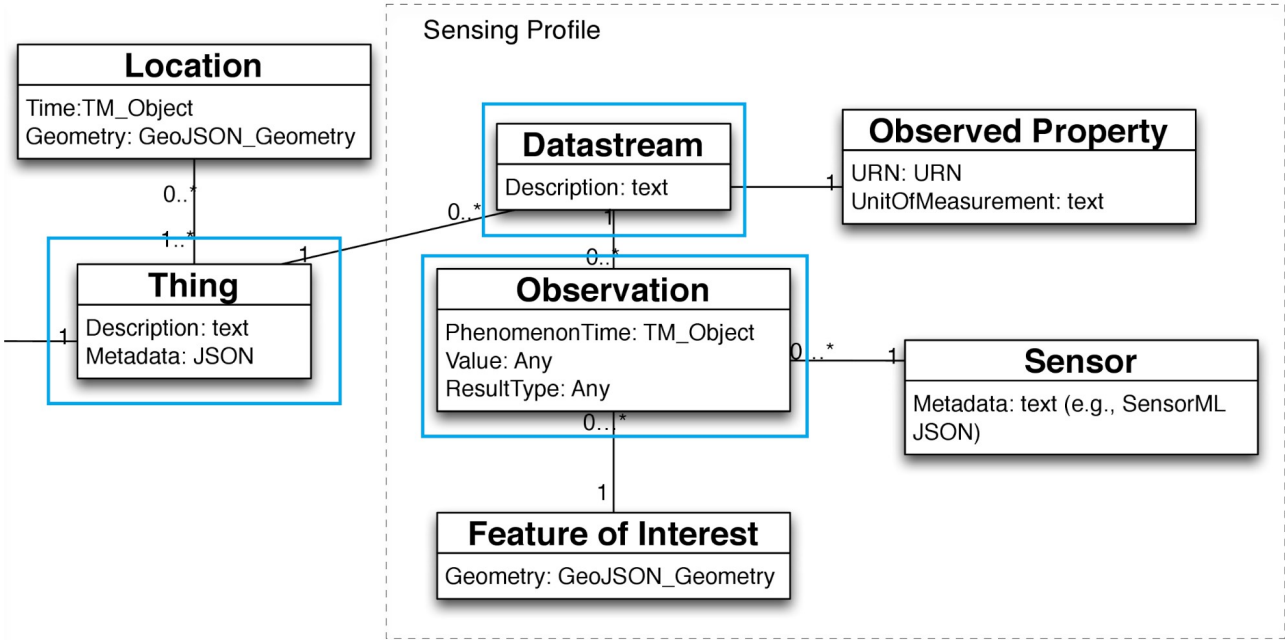

Figure 2: Core elements of OGC IoT data model and the Sensing Profile aspects, with the relevant attributes outlined in blue.

Use Case Implementation Status

There were several use cases identified and explained in the previous Technical Deliverables report [1], which have been implemented since then. As a reminder, the use cases are listed in Table 1 below, followed by an explanation of how they were implemented.

Table 1:Use cases identified and defined in [1]

1.1	Update database with formatted data
1.2	Format data according to OGC IoT standard
1.3	Uniquely identify sensors
1.4	Send data to Raspberry Pi

Use Case 1.1

The first use case identified for the LASS prototype was to post data to the data service, which is hosted at http://demo.student.geocens.ca:8080/SensorThings_V1.0/. Based on the components of the data model which were identified as relevant, this procedure was done in two main steps – initialization (corresponding to the CREATE function, or HTTP POST) and updating observations (corresponding to the UPDATE function). The first step creates the Thing with its related Datastream(s) and the second step creates Observations as they are read.

In the initialization step, a request is made to the data service to HTTP POST a “Thing”, effectively creating a new Thing to the database. Because of the data service implementation, the new Thing is automatically indexed based on how many Things are existing already. Once the Thing is created, a response is returned to the program which can be used to determine its location (URL). Before posting, the new Thing must include information regarding how many Datastreams belong to that Thing.

In the updating step, the Datastream of interest must be identified (which can be done using the URL returned from the initial request). Once it is identified, Observations that are assigned to the Datastream can be posted at regular intervals. All of the above steps were implemented using a free Python library called *requests*.

Use Case 1.2

The data format defined by the OGC SensorThings API for each entity follows the structure of JSON objects. Each of the different entity types have some attributes which are required and others which are optional. For the purposes of the project at this stage, this use case implementation was mostly hard coded. This can be considered as a first iteration of the use case solution.

At present, the Time and ResultValue properties are dynamically updated. The Time property is updated using the built-in time class and its isoformat() function and the ResultValue is updated based on the sensor reading. In order to test, sensor observations were simulated (as explained in the Status of Code section).

Use Case 1.3

Although the PIR sensor can be identified by the functions implemented in use case 1.1, we have yet to assign it an official identity. This will be done by consciously designing the electronic layout such that a sensor can be identified according to the GPIO pin it is connected to.

At the moment, this use case is not a great concern, since we are only working with one sensor and one datastream. Our goal is to create a data streaming process so that it can be scaled; in other words, if the process is fully functional with one sensor, it can easily be expanded across several more. We aim to expand our sensor inventory over the next month, and once we do so, this use case will be of more importance and will be completed.

Use Case 1.4

The fundamentals of progress in this area were easily completed with the help of studying several online projects posted by others, such as Matt Hawkins, [6] as a way to observe and understand the GPIO classes and functions. These libraries were previously unknown to the group members and as such had to be researched.

Several different methods of relating data from the PIR sensor to the Raspberry Pi via GPIO pins were tested and the most applicable libraries were chosen. These allow the sensor to receive data from it’s surroundings and relay it to the Raspberry Pi, where it can then be recorded, formatted and

posted to the network as per the other use cases. Currently we are able to make use of all the basic functions of the sensor.

Status of Code

At this point in time the group has created an overall python class that encompasses all of the use case goals (Table 2). Using this class, referred to as class “data”, the group has been able to:

- Record and format the datastream as required by use case 1.2
- Update the formatted data according to use case 1.1
- Send data to the raspberry Pi as specified by use case 1.4
- Use case 1.3 is included in 1.1 at the moment.

Table 2: UML Representation of “data” class

Class: data
ID: string
rootURI(): root URI *localtime():* time in local format *string():* string of time in required format *timeisoformat():* time in ISO 8601 format *initRequest():* creates and posts Thing to data service *sendObs(observation):* sends observation and time to data service

Although the use cases and the class “data” are functional, when seen from the perspective of actual implementation, they are still considered to be in a skeleton format. There are many checks and design modifications that must be added to make them robust which will be discussed in the risk section below.

Simulation File

When working with sensors and data streams, real-time testing is vital and yet can also be problematic when running the code. The sensors may not be ideally sensitive and real world conditions can interfere. It is for this reason we have created an additional python file that generates random values and ultimately simulates the PIR sensor, as well as accessing the “datetime” library in real time as the sensor would. This facilitated testing when posting data to the network as needed by use case 1.1. This simulation file will likely not be used after this point, now that use case 1.1 has been fully tested and is considered complete.

Website (Interface) Development

The website interface development has also made progress in implementing the use cases defined in the technical deliverables report. Table 3 outlines these use cases, with the details available on the project wiki [7].

Table 3: Use cases identified and defined for website

2.1	Assign location to a micro-controller
2.2	Configure a store layout
2.3	Get aisle traffic observations
2.4	Get facing observations
2.5	Get aisle traffic
2.6	See if products are faced
2.7	Show map UI
2.8	Log In/Authenticate
2.9	Become System Configurator

To facilitate the implementation of these use cases, the first necessary step was to construct a web-server to suit the needs of the project. Due to the web-server being behind the scenes, with actors not interacting with it directly, the work done cannot be assigned to a single use case. Rather, it forms the framework for all of them.

Work on specific use cases has begun in the form of the Store Layout Creator (use case 2.2). The Store Layout Creator is a web page that allows users to make a digital representation of a physical store. Along with use case 2.2, this work also contributes to the following use cases:

> **2.1** – With the store layout configured, a microcontroller can be assigned to a shelf, which has a location. The ability to edit shelf attributes in the store layout accomplishes this use case, as this allows the microcontroller to be mapped to a shelf. At the time of writing, this is done by supplying the URL of the microcontroller on the OGC SensorThings API.
>
> **2.5 & 2.6** – Similar to assigning a location to a microcontroller, sensors can be assigned to sections within shelves. This gives them a location on the map UI, which accounts for part of these use cases. The remaining work will be to visualize observations on the map.
>
> **2.7** – Since the store layout creator visualizes the map as it is being built, the majority of work in this use case is done. The remaining work will be to render the visualization on a separate page, which can be accomplished once the store layout is saved to a database.

Work has also began on use cases 2.3 & 2.4, in the form of a demo application. The purpose of the demo is to get and display data from the OGC SensorThings API server. While the demo does not specifically provide any functionality for the final product of our website, the code can be easily tweaked and re-used after demonstration. The details of the work completed on the website interface is explained in the following sections.

Web-server Development

In order to effectively serve and store content for the proposed website, a small server needed to be created. The purpose of such a server was twofold: we needed something that could serve dynamic pages in a clear-cut fashion, and likewise needed a way to respond to user requests from the server (such as in the case of adding sensors or shelves to profile). Thus, development of a simple web-server began. The requirements of such a server were:

1. Must be able to load content dynamically – for this reason, a templating language or specification for HTML content was desired
2. The web-server should ideally be capable of connecting to a database of some kind (Redis, MongoDB, or PostgreSQL) in order to store user data and handle log in / register requests.
3. The code for the server should be split up in a modular fashion. In particular, a model-view-controller or MVC approach is preferred, to make development and deployment rapid and easily iterable.

Working off of the requirements above, a simple web-server was implemented using the Node.js runtime [8] while using the Express.js framework [9]. Using Express.js made development of the server incredibly easy, and allowed the group to maximize the amount of time spent on the functionality of the website itself, while minimizing time wasted trying to serve content. To satisfy the first requirement above, Mustache templating [10] was likewise used, for its simplicity and the ease of which it could be integrated into Express.js. Done in this way, the potential learning curve of various web technologies was reduced.

Of particular note is that there is not yet any specific support for a database, nor is there currently any way to store user data. A few approaches were researched, such as implementing a PostgreSQL database to return JSON fields, or even running a Redis server (NoSQL) to store user data. The primary roadblock was that neither were easy to set up while the extent of the user data required to be stored was unknown. While this could have been implemented multiple different ways, this particular subject has been avoided for the time being in favour of placing more effort towards developing the basic functionality of the website. Given more time in the future, this avenue will be explored more thoroughly. Overall this was determined to be a minor risk given that the primary functionality of the website will not differ significantly as a single-user system vs. as a multi-user system.

Lastly, in an attempt to prevent clutter of code within the ENGO500 repository [11] on Github, the group has decided to split the website and server code from the main repository. All of the website functionality and code can now be found in the ENGO500-web-server repository [12], likewise available on Github.

Store Layout Creator

Due to differences in physical layouts between stores, the web application required a means to create a digital representation of a specific physical space. Not only does this representation need to make sense in terms of data, but also requires a visual component. Once the store layout is configured, it will be used to display observations such as an activated PIR motion sensor or a shelf that is low on stock. To accomplish this, a combination of JavaScript, jQuery, and D3.js were used. JavaScript and jQuery were used to create and manipulate the objects that define the store layout, as well as organize them into an easy to configure manner. D3.js was used to visualize these objects in a way that reflects the physical world. Together they make a coherent system which will facilitate the initial set-up of the LASS in a real-world space. The creation of such a system went through 2 major iterations to reach the state it is at the time of writing. The following section details the two iterations by providing background information, design choices, and how the systems work.

Technologies Used

jQuery

jQuery is a JavaScript library that simplifies HTML document manipulation, event handling,

animation and Ajax. This allows for the creation of dynamic web pages in a much simpler manner than that of using straight JavaScript. Used by 57.7% of all websites, it has the largest market share of JavaScript libraries at 93.1% [13]. It was chosen due to the library being free, open source and very prevalent. The API is well documented and there are many tutorials and help references available.

jQuery UI

jQuery UI is a set of user interface elements and themes that build on top of jQuery's feature set. For the store layout creator, the accordion widget was used to house an editable representation of the underlying objects. This widget is a collapsible menu which can hold a lot of content while minimizing scrolling.

D3.js

D3.js is also a JavaScript library, but it exists for a different reason than jQuery. D3's main use is for visualizing data within web pages. It does this by generating and styling SVG graphics which can include transitions and other dynamic effects. It was chosen for the project due to it's ability to portray GIS data while utilizing web standards all while being free and open source.

Jeditable

Jeditable is a jQuery plugin that allows in-line text editing. It was chosen due to it providing the exact behaviour needed for the store layout creator. It is provided with an MIT license.

Underlying Data Structure

While the exact data structure has yet to be determined, it has been implemented in such a way that editing or adding to it is trivial. As it exists at the time of writing it is as follows:

```
Shelf:{
      shelfName: ,
      notes: ,
      rpUUID: ,
      sections:[{
            sectionName: ,
            displayId: ,
            displayColor: ,
            pirURL: ,
            pintURL: ,
      }],
}
```

An array of shelves is initialized when the store layout creator is opened, and shelves & sections are added to it as needed. Shelves serve as a container for sections, and also have a number of attributes such as a name, and an attribute used to match the shelf to a raspberry pi. Sections are for organizing products within the shelf. An example of this organization could be a shelf holding snack food, with sections for chips, nuts, cookies, etc.

Iteration 1: Using Solely D3.js

When the task was initially taken on, an attempt was made to provide a solution using only D3.js and JavaScript. This choice was mainly made due to unfamiliarity with D3.js. It was known that D3.js would be able to provide the level of manipulation that the team wanted, but the complexity

of providing such a solution was not known. The details of this implementation follow.

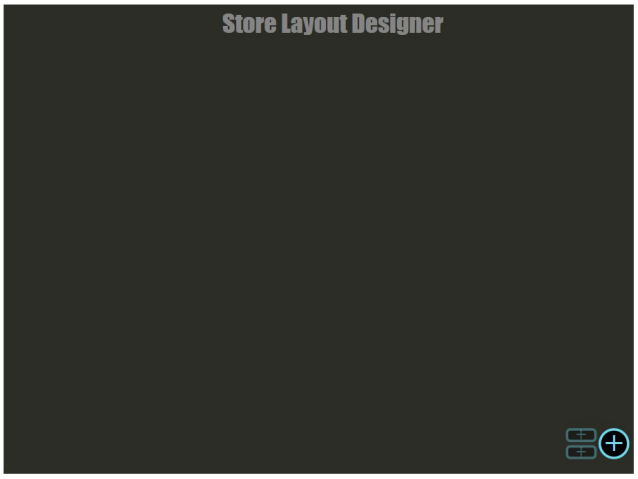

Figure 3: Greeting screen

Figure 3 above shows the initial look of the first iteration layout creator. Users were greeted with a mostly blank screen with a single enabled button in the bottom left. Clicking this button would add a shelf to the screen, as seen in Figure 4:

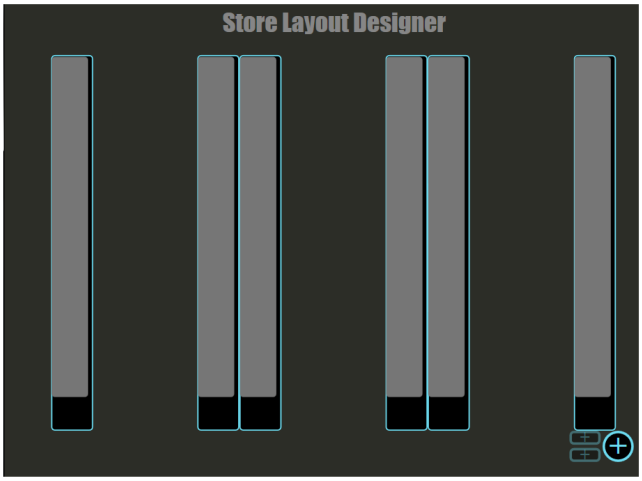

Figure 4: Shelves Added

The black rectangles with the blue outline represent shelves, and the grey rectangles represent a section within that shelf. The spacing of these rectangles is not correct for a finished product, but served as a prototype due to the design being abandoned shortly after. At this point users were intended to click on a shelf, which would highlight it as shown in Figure 5.

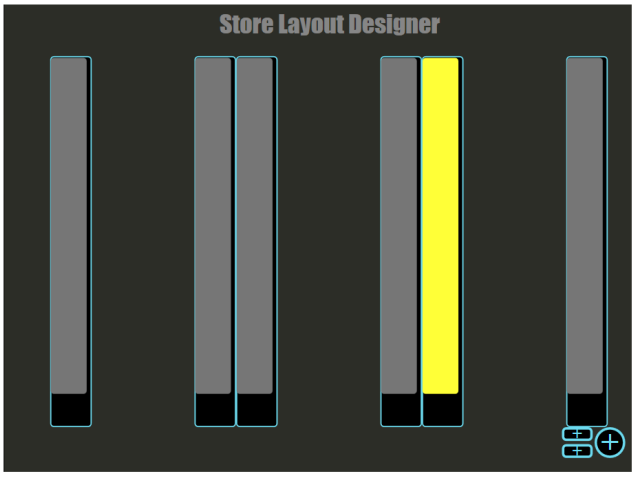

Figure 5: Section selected

Additional controls become activated as seen in the bottom right hand corner. These two stacked buttons allow the user to add additional sections to the shelf. Selecting a shelf was also meant to bring up controls for configuring that section's attributes. This step of the implementation is ultimately what changed the course of the development to include a way of manipulating the layout outside of an SVG interface.

The last piece of functionality implemented was that the existing section would resize to half of the available space within a shelf when one of the add section buttons was clicked (see Figure 6). The next step would have been to generate a new section and display it in the empty space.

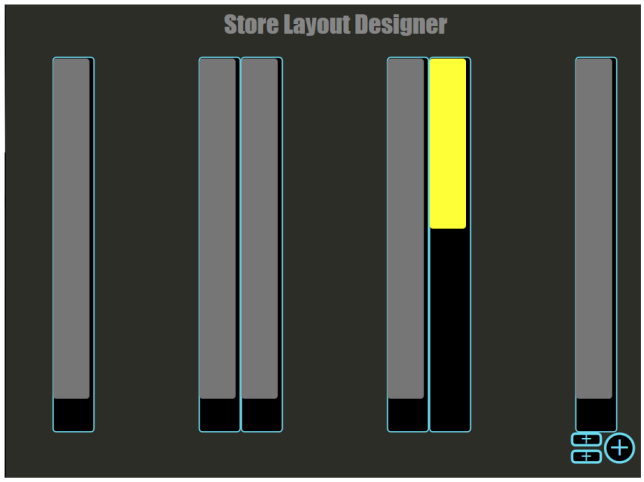

Figure 6: Section Split

Along with the aforementioned issues with this iteration, another design issue is that there is very little guidance given to the user. These qualms prompted the second iteration design.

Iteration 2: jQuery and D3.js

One of the first things tackled by the design of the second iteration was the lack of direction given to the user. This was done by simply designing a better HTML layout. Figure Error: Reference source not found shows the store layout screen as it appears before the user interacts with it. Already there is improvement in that there are some text instructions and the '+ Shelf' button is labeled rather than purely graphical as seen in iteration 1. The page was structured into a 30/70 split to accommodate both the manipulation and visualization elements.

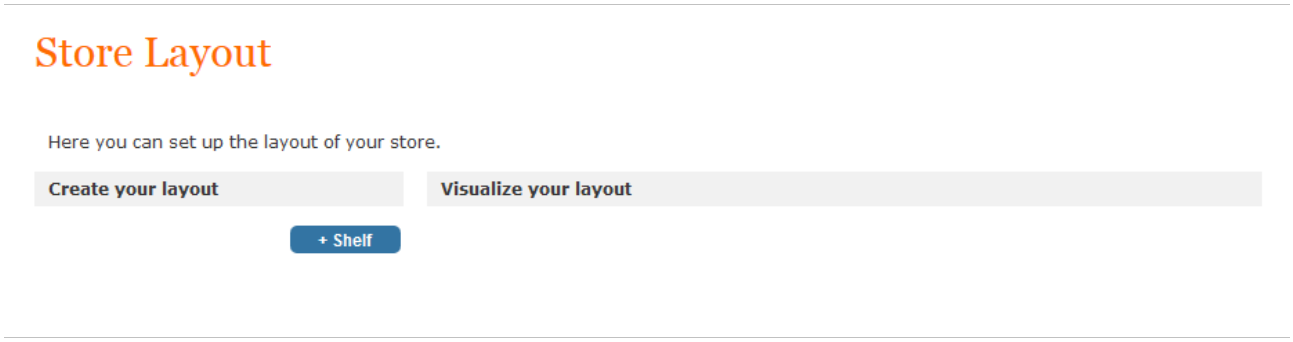

Figure 7: Second greeting screen, using jQuery and D3.jjs

Clicking on the '+ Shelf' button allows users to add a blank shelf element to the screen. This is represented both by an accordion element in the left pane, and an SVG element in the right pane, seen in Figure 7. Multiple shelves can be added, and the spacing of the shelves will adjust automatically.

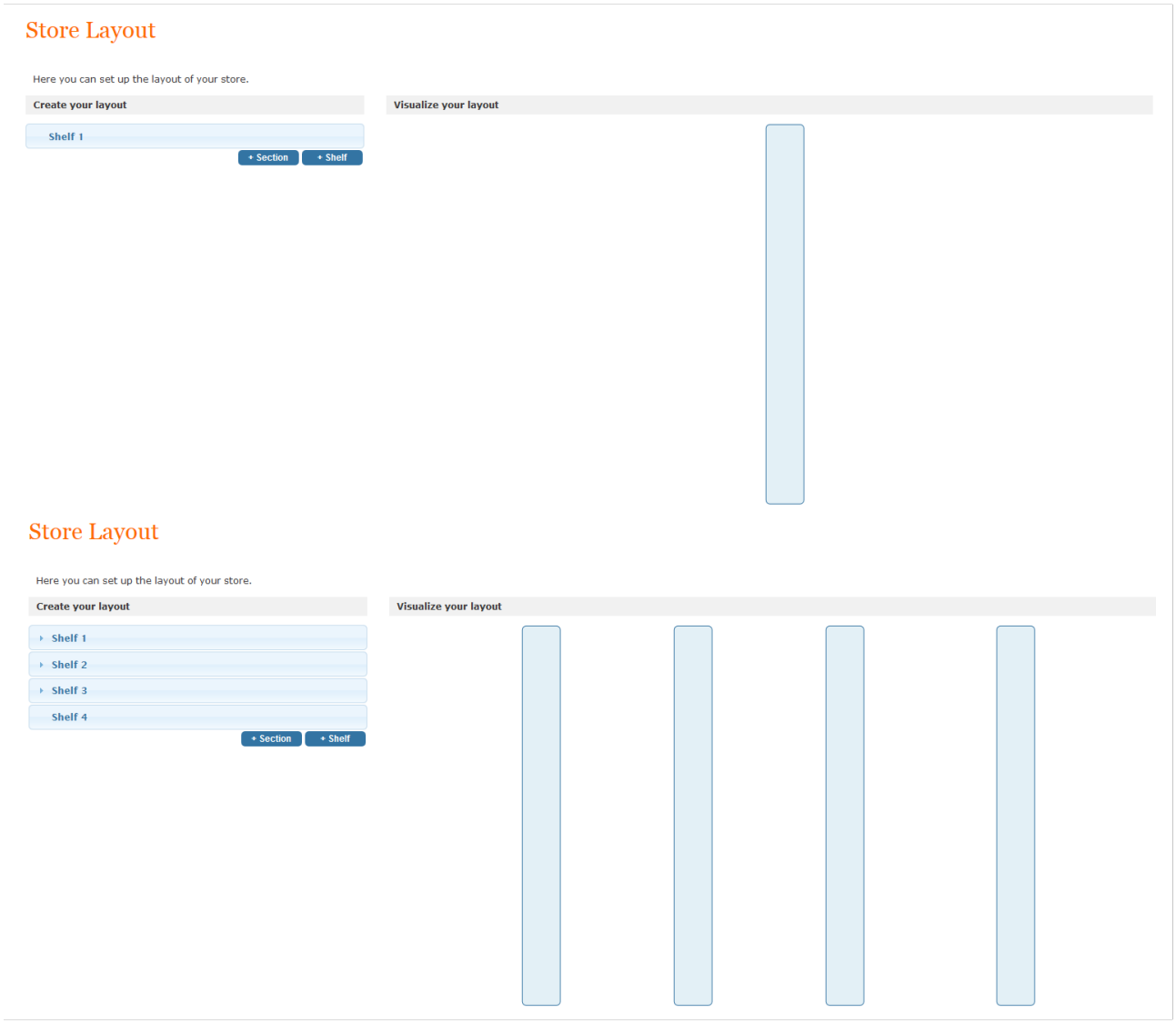

Figure 7: Single and multiple shelves added

Opening a shelf accordion element in the left pane shows the existence of the shelf attributes (Figure 8). When the shelf is first created, this attribute section is automatically generated. With a shelf selected (accordion is open) a user can then add sections to the shelf using the '+ Section' button. Similar to adding shelves, both an accordion element and an SVG element will be generated which can also be seen in Figure 8:

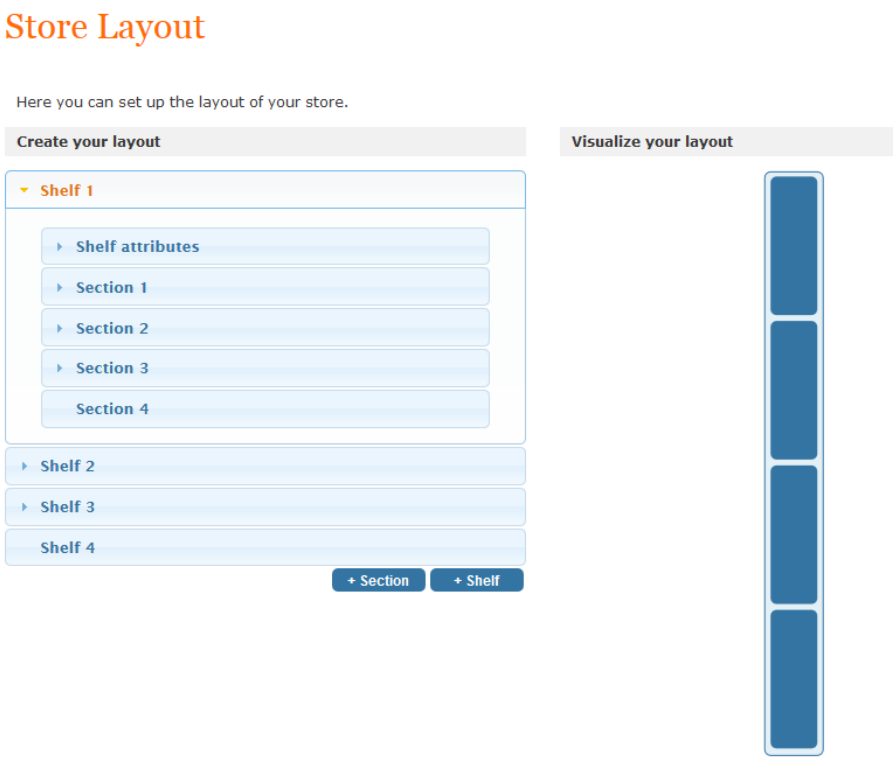

Figure 8: Shelf with attributes and four sections

If a user clicks to open one of these sections, they are shown the attributes belonging to either a section or a shelf. When the object is first created, these variables are initialized but do not hold a value. They are displayed with the message 'Click to edit' which informs the user how they should interact with the element. This can be seen in Figure 9, below:

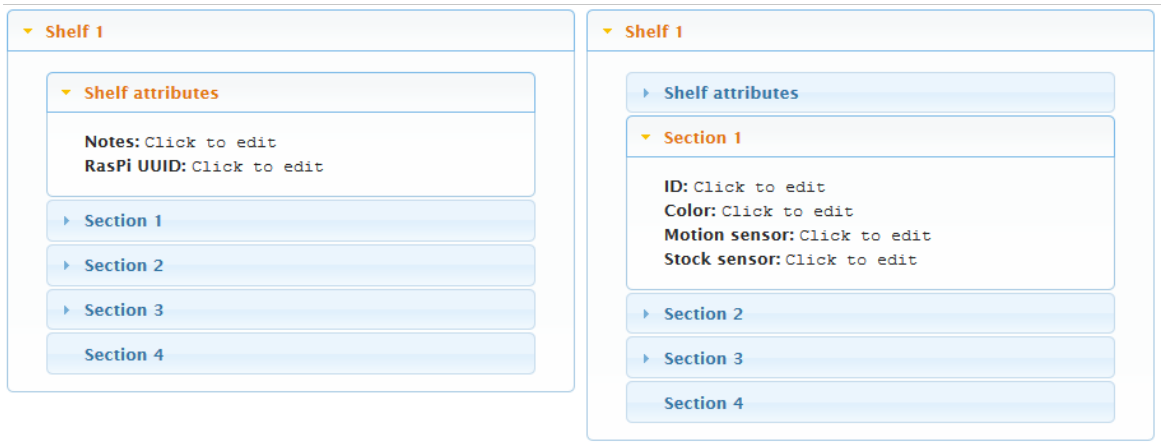

Figure 9:Editable attributes within accordian

Clicking on one of them will enter an editing mode. The user can save their input by pressing enter, or cancel by pressing escape or clicking outside of the text box. Once saved, the underlying object

will be updated to retain the input value. Figure 10 shows this process.

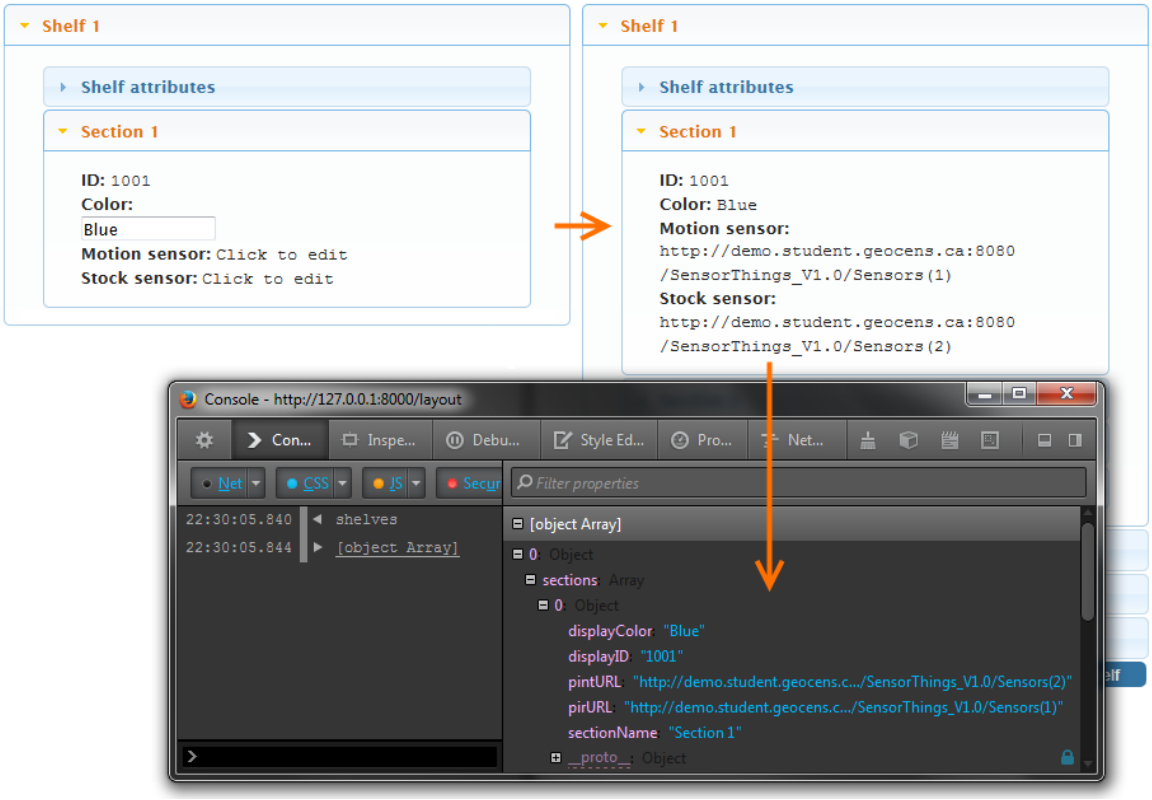

Figure 10: Editing attributes and reflected change within data structure

When fully configured, the user is left with a layout like the one shown in Figure 11. While this system does not yet contain the full feature set, the core functionality exists. Additional enhancements will be detailed in the schedule section.

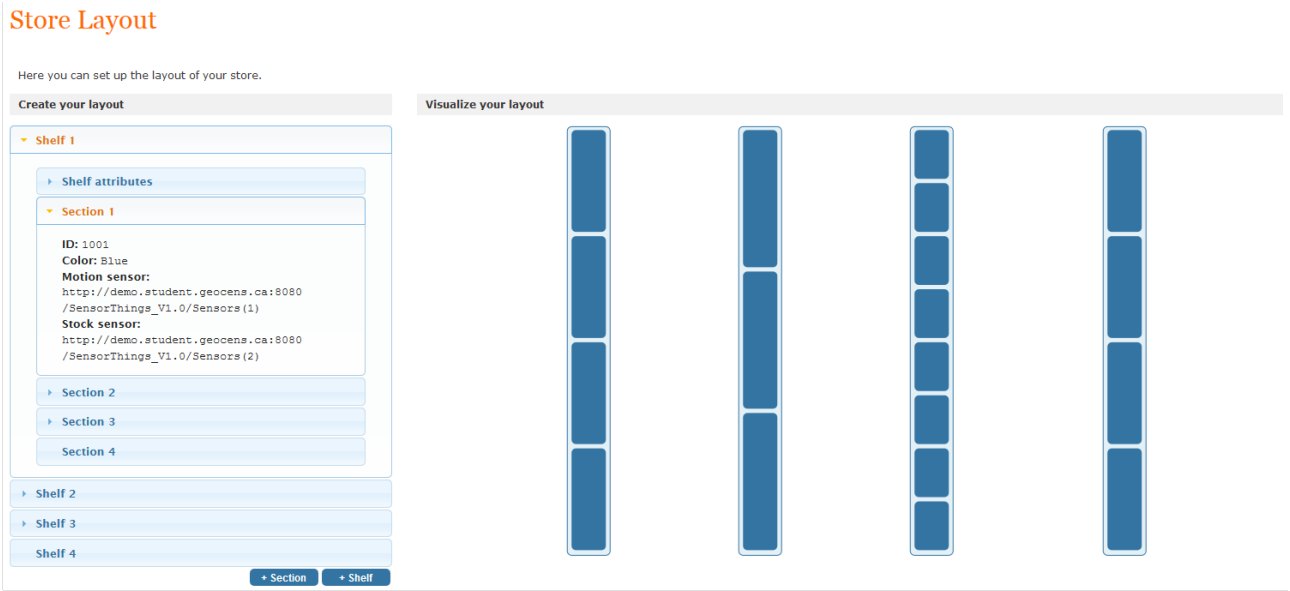

Figure 11: Example of a store layout

Demo – Data Relay

As noted in Use Cases 2.3 and 2.4, we need to be capable of pulling data from the OGC IoT SensorThings API dynamically in real time. As a partial exercise to show off this functionality, and likewise explore the data on the server in more detail, a small demo was created. This demo pulls all of the data for each 'Thing' on the server, and organizes it into an interactive force-graph using D3.js. An example screenshot of this demo can be seen in Figure 12 below:

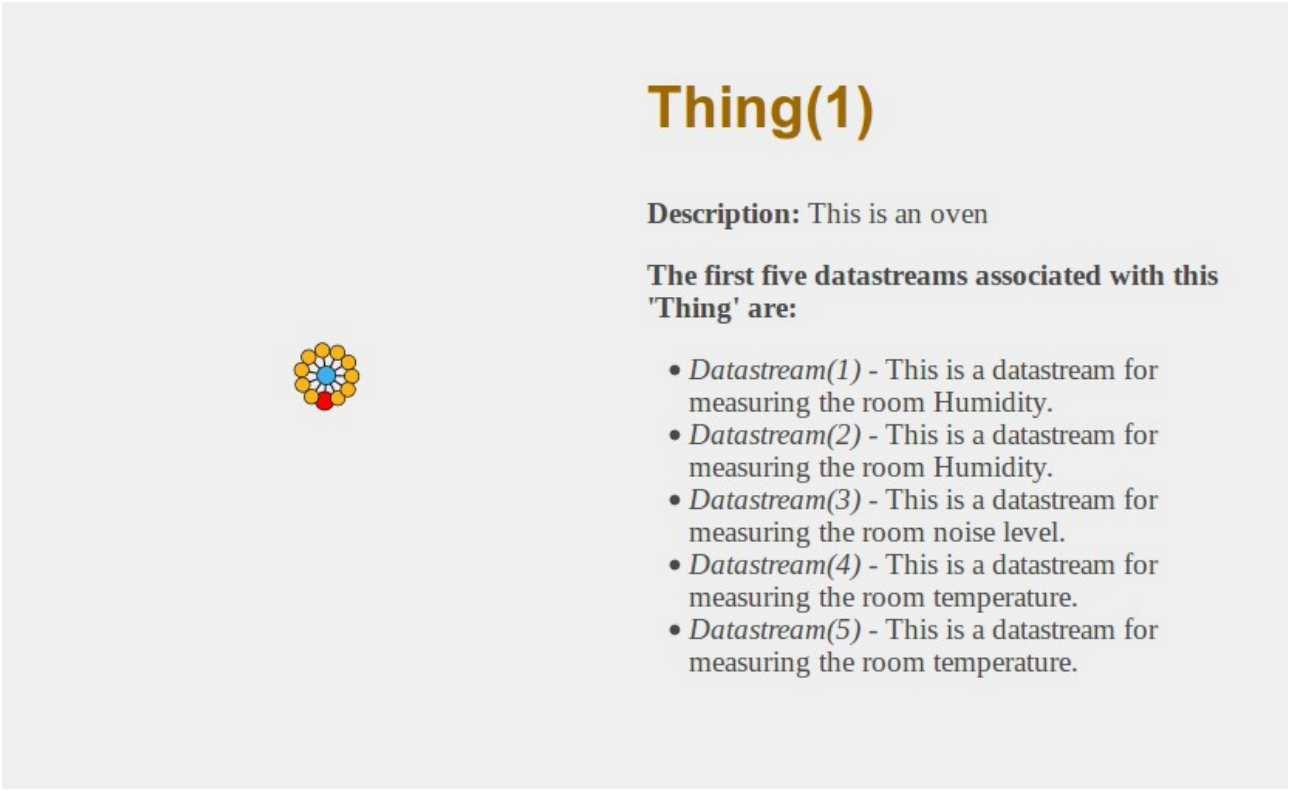

Figure 12: Example of force-graph demo pulling data dynamically from server and displaying it alongside the graph

While the above demo code doesn't specify the exact way that such code will be used in a production setting, it does provide an interesting visualization of the uploaded objects into the API. What's more, since this demo loads dynamically when the page is loaded, the visualization changes as objects and observations are added to the server. The group expects the transition from the above code to production code to be trivial in nature, once the specific functionality is required by other pages / modules.

Schedules, Risks, and Status

Schedule

Since the previous technical deliverables report, much effort has been put into finalizing our use-case definitions, and likewise working towards implementation of our individual use-cases. Overall, we have almost entirely (~70% - 80%) implemented most of our use-cases with regards to our prototype development, and have for the most part implemented four of our nine (~40%) proposed use-cases with regards to the website.

Major roadblocks regarding use-case implementations can be attributed to two primary factors: delays with the shipping of our hardware / sensors, and difficulties in determining the simplest way to implement basic authentication and database storage within the web-server. While it appears that

few of the use-cases have been implemented on the web-server side, it is important to note the distinction that many of the use-cases depend on the primary functionality provided by use-cases 2.2 and 2.7, which have currently received the most attention to date. Moreover, much of the code regarding many of our use-cases can be tweaked and re-used in other use-cases as well (e.g. displaying the map UI is necessary in both 2.2 and 2.7, and largely uses the exact same data elements). To see a full layout of how we expect these delays and roadblocks affecting the overall progress of our final project deliverables, see the modified Gantt chart shown in Figure 13.

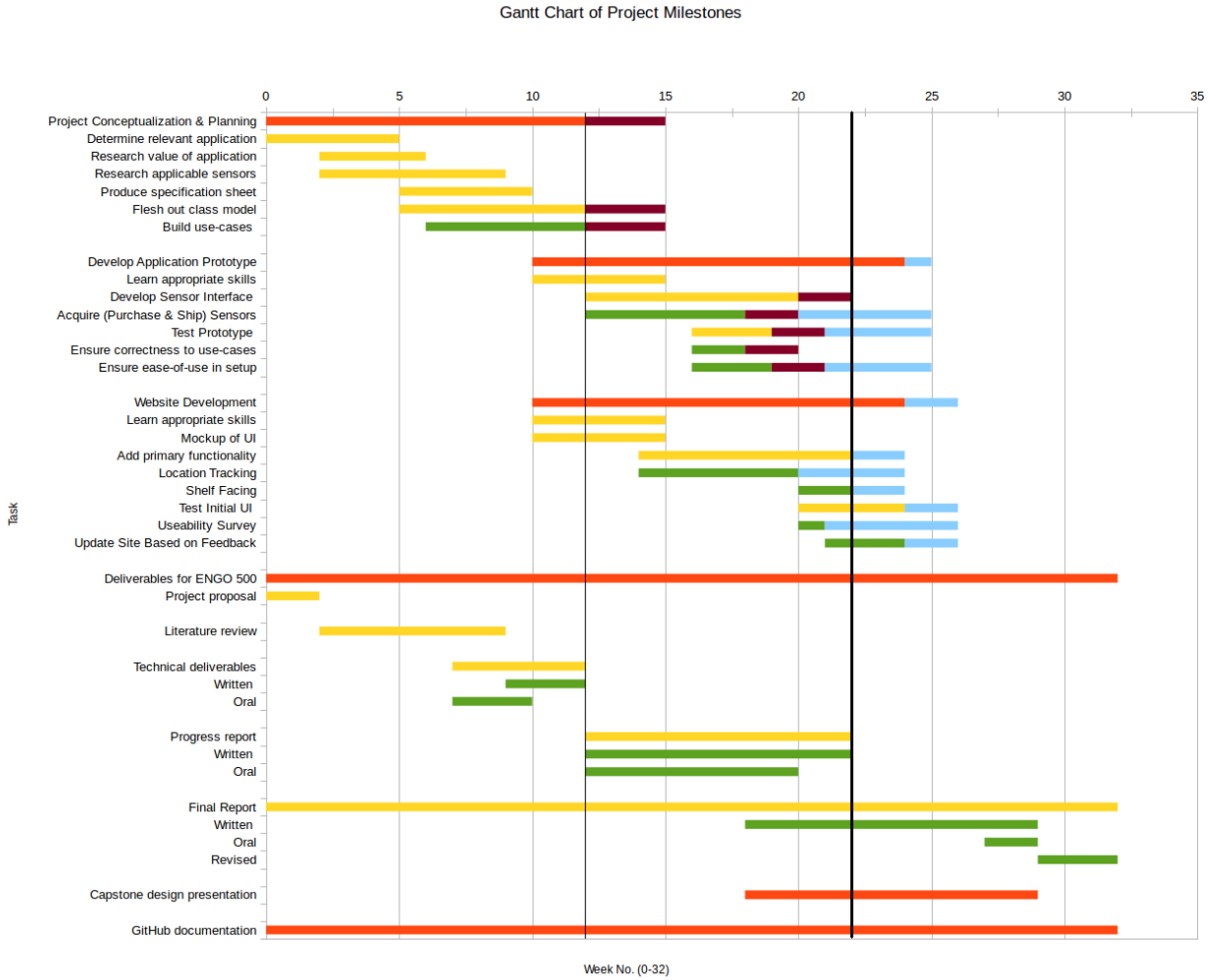

Figure 13: Modified Gantt chart. Thin bar represents progress as of Technical Deliverables report, thick bar represents progress as of this report. Blue areas are potential delays in elements of the project as discussed in the risks section below

As seen in the Gantt chart above, the largest effects relate to post-development and usability testing of the prototype and website. Given the iterative approach to our development, bugs have been dealt with as they appear, and most of our completed functionality is bug-free. A more detailed outline of risks can be found in the subsequent Risks section.

Future Work

Given the schedules and progress outlined above, the prototype development will aim to accomplish the following goals in the coming weeks/months:

LASS Prototype Development

- Add user-friendly features, such as having an on/off switch and status indicator LEDs
- Modify data collection program to take real data from a PIR sensor once new sensors arrive. The group also has the option of attempting to correct the spastic-nature of our current sensor
- Generalize class “data” for more than one datastream or sensor type
- Include ID tag specific to each Raspberry Pi. This will most likely be hardcoded
- Finish implementing use-cases completely (final 20%-30%)
- Develop and finalize physical prototype (requires that sensors be shipped)

Website (Interface) Development

- Enhance interactiveness of shelf layout (tooltips, colours and IDs, click section to open appropriate accordion)
- Make editing easier (if you select a raspberry pi for the shelf, maybe make the sensors belonging to the Raspberry Pi that appear in a dropdown for the sections, dropdown for colors, etc).
- Change spacing of shelves to reflect real world (2 grouped)
- Fix known bugs
- Save configuration to database
- Add ability to load configuration (user data) from database
- Ability to delete in layout editor so we adhere to CRUD design
- Finish implementing use-cases

Risks

Several risks that could become a threat to the proposed timeline for the remainder of our project are considered below. Primarily, there is always a chance that the sensors we have ordered through our advisor may arrive later than expected. This risk is possibly the most threatening since is it not completely within our control to prevent. However, our hardware is currently shipping and is expected to arrive within the next week or two, which still allows for plenty of time to test and work with the hardware provided that they do not arrive late.

Furthermore, there is a potential risk that the ordered sensors may not function as described upon time of purchase. This has already caused some concern among the group, since the PIR sensor we are currently working with is proving to be over sensitive, which we did not expect. Several solutions are being considered to counteract its oversensitivity. One possibility to mitigate is a redesign of our sensor positioning, such as placing it on the ceiling to remove vibrations caused by the environment of the shelf face.

Another risk to consider lies in the robustness of the connectivity. Connection to the network is necessary to pass the datastream to the website, and if a connection cannot be made for any reason, the user should be notified of the problem. Currently there is no alert for this sort of error. The group will create checks that will report such an error, although it may not be able to identify and account for the vast variety of potential problems. However, the priority is not fix or identify the connection error, but merely to alert the user of a problem. With this in mind, this risk is

approachable in our remaining time frame, and does not pose much of a threat. Moreover, given that in production the system would likewise be under a wired ethernet connection or a closed wi-fi network, network stability becomes less and less of a problem.

With regards to the website development, the biggest risk involved is in not fully implementing all of the functionality of the website. Specifically, with regards to a multi-user system, this may not be possible within the time-frame. More research currently needs to be put in to explore this topic in more depth, and find the optimal solution that can be implemented within time. On the grand scale of the project, this risk isn't too important, as a single user system with the remainder of the basic functionality implemented would be sufficient as a first iteration of the project requirements.

As mentioned previously in the Schedules section, because of delays in other areas of development, some of our usability testing may not be completed, or may be delayed significantly. Given the simple nature of the website functionality and our iterative development process, however, we believe that this will not present significant hurdles in terms of usability for our final site, as we are performing iterative testing and filing bug reports in code as it is being developed.

Our final concern regards downloading and displaying massive bouts of data from the server. Because the server has refreshed itself numerous times since we have started the project, we have not had any chance to download and test the speed of the server given a large data set. This may or may not present problems when we aim to show a map of observations over a store layout, as we do not know how reliable or easy it will be to retrieve and handle large quantities of data from the server.

Risk Mitigation

In order to mitigate the risks outlined above, some changes to the group focus and our methods have been considered:

1. Much of the prototype development is ahead of schedule, despite the sensors shipping late. For this reason, we believe it to be in our best interests to move some of the man-power off of the prototype development and move them to assist with building the basic functionality for the website.
2. As previously mentioned, changing the sensor configuration from the shelf to perhaps on the roof or another area has the potential to mitigate some of the issues experienced with the PIR sensor sensitivity.
3. Future focus will be more oriented towards developing the basic functionality of the website. With #1 above, the group believes that we can mitigate any fears or risks that we may encounter regarding not being able to implement certain features, such as a proper login / authentication system.

Conclusion

The project has progressed considerably since the Technical Deliverables Report was written and submitted. Both components of the project (shelf prototype and web interface) have been significantly developed, with various sample applications/programs having been created to demonstrate functionality.

In terms of shelf prototype development, all four use cases previously identified have been implemented to usable, albeit not robust, functionality. Use case 1.1, which addressed uploading data to the data service, was a key link between the two sides of the project. The Raspberry Pi can now successfully register itself as a Thing in the data service, and can post observations to a designated datastream. Knowledge from this process can be extended to more sensors when they

arrive. The other three use cases (1.2-1.4) primarily address functionality which occurs on the Raspberry Pi locally, and have been achieved with some manual effort. For example, the Raspberry Pi can identify a sensor based on which GPIO pin it is attached to, but this should be established by the device creator (and not modified by the client). Reading the data from the sensor was achieved by using the GPIO pin library, and formatting the data according to the OGC IoT SensorThings API was accomplished based on examples in the OGC IoT Interactive SDK. From here, the work will be extended to include different sensors and to increase the robustness in implementation.

As for the web interface side of the project, progress had been made in both creating a web-server and also implementing use cases. The web-server technologies chosen were a node.js web-server using an express.js framework. These were chosen for ease of implementation as well as flexibility between client side and server side code. The remaining work in this area will be to implement a database to store user settings and content. In terms of implementing use cases, significant progress has been made on a Store Layout Creator (use case 2.2) and the use cases it supports (2.1, 2.5 – 2.7). At this time, it is possible for a user to create a representation of their physical store in a dynamic environment that allows editing of attributes and also provides a visualization of the underlying objects. The team has also implemented as much of use cases 2.1, 2.5 – 2.7 as possible, with most of the remaining work waiting on implementing a database to save settings. Additionally, use cases relating to getting data from the server (2.3 & 2.4) have progressed as the team is able to get and display information from the OGC SensorThings API. Outstanding use cases on the web interface include those dealing with user management, such as being able to log in with different users. These have been deemed low priority in favor of getting a single user system working with full functionality.

Based on the progress made on both the sensor prototype and the web interface, the team is confident that we will be able to produce a working implementation of our design concept which is compliant with the original project requirements.

References:

[1] 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.

[2] Margaret Rouse, “What is the Internet of Things (IoT)?,” *whatis.com*, Jul-2013. [Online]. Available: http://whatis.techtarget.com/definition/Internet-of-Things. [Accessed: 05-Dec-2013].

[3] “OGC Standards and Supporting Documents,” *OGC Standards and Supporting Documents*. [Online]. Available: http://www.opengeospatial.org/standards. [Accessed: 06-Dec-2013].

[4] Adafruit, “Hardware | Adafruit’s Raspberry Pi Lesson 12. Sensing Movement | Adafruit Learning System,” *Hardware | Adafruit’s Raspberry Pi Lesson 12. Sensing Movement | Adafruit Learning System*. [Online]. Available: http://learn.adafruit.com/adafruits-raspberry-pi-lesson-12-sensing-movement/hardware. [Accessed: 13-Feb-2014].

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

[6] Matt Hawkins, “Cheap PIR Sensors and the Raspberry Pi - Part 1 | Raspberry Pi Spy,” *Cheap PIR Sensors and the Raspberry Pi - Part 1 | Raspberry Pi Spy*, 23-Jan-2013. [Online]. Available: http://www.raspberrypi-spy.co.uk/2013/01/cheap-pir-sensors-and-the-raspberry-pi-part-1/. [Accessed: 13-Feb-2014].

[7] 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].

[8] “node.js,” *node.js*. [Online]. Available: http://nodejs.org. [Accessed: 13-Feb-2014].

[9] “Express - node.js web application framework,” *Express - node.js web application framework*. [Online]. Available: http://expressjs.com/. [Accessed: 13-Feb-2014].

[10] “{{ mustache }},” *{{ mustache }}*. [Online]. Available: http://mustache.github.io. [Accessed: 13-Feb-2014].

[11] Jeremy Steward, “ThatGeoGuy/ENGO500,” *ThatGeoGuy/ENGO500*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500. [Accessed: 05-Dec-2013].

[12] Jeremy Steward, “ThatGeoGuy/ENGO500-web-server,” *ThatGeoGuy/ENGO500-web-server*. [Online]. Available: https://github.com/ThatGeoGuy/ENGO500-web-server. [Accessed: 13-Feb-2014].

[13] “Usage Statistics and Market Share of JavaScript Libraries for Websites, February 2014,” *Usage Statistics and Market Share of JavaScript Libraries for Websites, February 2014*. [Online]. Available: http://w3techs.com/technologies/overview/javascript_library/all. [Accessed: 13-Feb-2014].