

Balloon-Assisted Microgravity

Group 31

Andrew Geltz, Luis O'Donnell and Priscilla Ryan

Sponsored by Florida Space Institute

Fall 2017

Table of Contents

Executive Summary	4
Proposed Solution	5
Broader Impacts	6
Personal Motivations	7
Existing Alternatives	8
Project Requirements	10
Design Considerations	11
Sensors	11
Communications	12
Hardware	14
Software	19
Implementation Details	43
Sensor Interface and Communications.....	43
Database	46
Front End Display	61
Initial Server Setup	70
Getting Started	80
Project Milestones	83

Prototype and Evaluation Plan	84
Budget and Funding	86
References	87
Appendices	88
Copyright Permissions	88
Datasheets	94

Executive Summary

The purpose of this project is to create an efficient system for testing microgravity experiments. We are continuing the work of the Mechanical Engineering students who completed their final semester in last Fall. The basic idea of this project is to take an aerodynamic capsule with an experimental payload up to 100,000 feet using a high-altitude balloon with a winch system, release the winch and collect data as the capsule descends. The capsule will remain tethered to the balloon but should fall at speeds to simulate microgravity. When the capsule reaches a certain height, the tethering system should slow the descent of the capsule to a stop and retract the tether to reset for the next test.

This team will be working on the ground station aspect of this project. This ground station should display relevant sensor data from the capsule as well as being able to issue commands to the capsule and display any messages generated by the capsule. We worked with the existing design of the capsule and platform systems and collaborated with the Mechanical Engineering teams to try to implement the communication capability of the ground control station.

Proposed Solution

Our proposed solution is to establish a rapidly updating web interface to show the progress of the experiments and the sensor output. The capsule will transmit its readings, over RF, to the raspberry pi, which will then upload these readings to the database. A web server will then interpret the data in the database and serve it to clients over the internet (or locally). The control flow also works the other direction with the interface's commands. The user would send the command to the web server (ie to start or stop a test) which would transmit the command to the Raspberry Pi. The Raspberry Pi will then forward the command to the capsule over the RF communication channel.

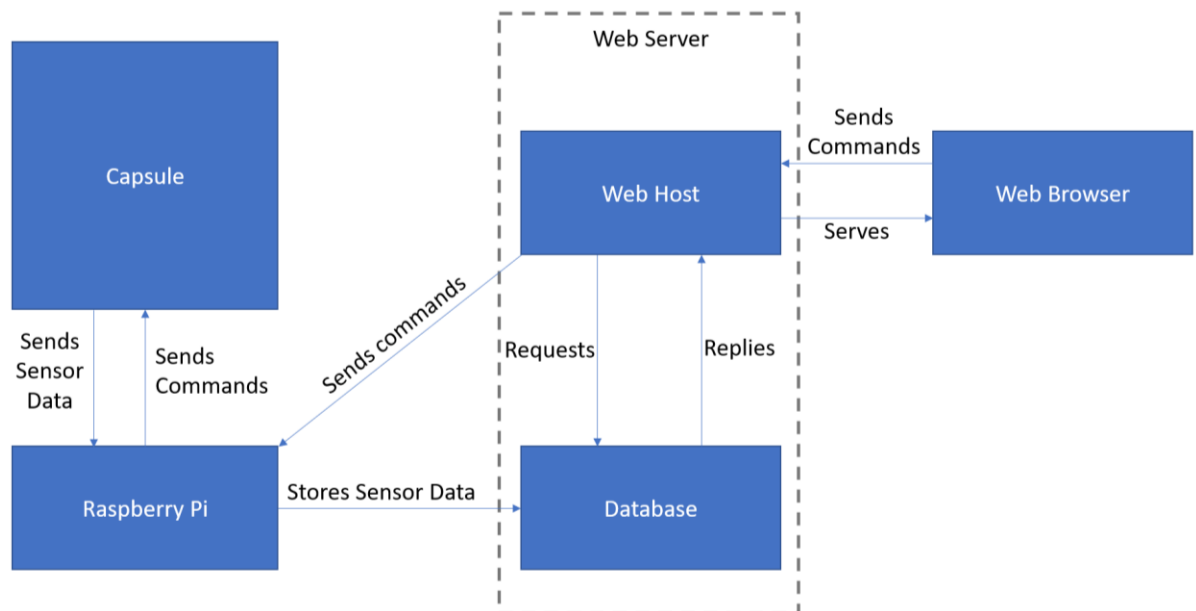


Figure 1 Proposed Solution

Broader Impacts

This project will have a large impact on the way that reduced gravity experiments are conducted. There are a few current systems that work but are not optimal in terms of cost and performance. One current option is to send the experiments to the ISS (International Space Station) which is expensive and time consuming. Another approach is using drop towers. Drop towers are not optimal because the height is usually limited and the payload may not experience true microgravity in the drop. The aim of this project is to provide 9 to 25 second tests in a reusable method that isn't as time consuming or as expensive as other methods.

Reduced gravity experiments are used to test systems for space or other conditions that would require extreme gravitational changes. Making these tests more affordable and faster would reduce turnaround time when developing new technologies for space and aerospace.

Personal Motivations

Andrew Geltz

This project was a very interesting proposition for me because I was familiar with the current state of microgravity experiments. The current systems are outdated, expensive or do not meet length requirements for many applications. I have a background in electronics from my participation with the robotics team at my high school. I wanted to work on a project that I could learn up and coming technologies. Anything that will help science as whole is something I am excited about.

Luis O'Donnell

I have had an interest in Space and Earth science for a long time. During my time in high school I have done science projects regarding the Earth itself. These projects focused on the Earth's magnetic field and its fluctuations. For these projects I had to learn a great deal about the Earth itself. Having to learn more about the Earth was interesting and honestly fun. Since high school I've been meaning to take a class on Earth and space sciences but I have actually yet to do this. Now that I get to finally be able to work on a project that is related to this topic I'm extremely excited. I am looking forward to be able to learn new things as well as use my computer science skills on this project.

Priscilla Ryan

I've always been captured by science and its incredible advances in many areas, but none more so than that of space exploration. Space is indeed the final frontier, so getting a head start on research now is important in the long run. And because of the nature of space, microgravity experiments in particular are crucial when looking at proper research and exploration. Therefore, optimizing their conduction is something that I believe is very important to focus on, and is thus a significant source of motivation for me to work on this project.

Existing Alternatives

The basic idea behind this project is to make the use of a microgravity testing environment become readily available to more people. The idea of using a balloon up thousands of feet into the air to simulate this environment, is to make it so these types of experiments can actually be performed more often. Other methods of being able to perform experiments under a microgravity environment do exist but are much harder to obtain access to.

One such method of being able to perform these experiments is on a parabolic flight. Parabolic flights, simulate a microgravity environment on the flight. During these flights, at usually an altitude of 20,000 feet, a microgravity environment is created by rapid descents. Weight will become 1.8 times greater during the flight on ascents, but for a short period during the parabolic flight will experience zero gravity. Due to a change in trajectory of the flight to resemble a parabola, perceived weight goes from 1.8 times greater than normal to zero gravity for approximately 20 seconds. This is the time period in which zero-gravity is present during the flight. The only problem is that the zero-gravity environment only lasts for 20 seconds. Even though it is possible to perform these experiments, being able to have access a parabolic flight often is quite uncommon.

NASA has a program called the Flight Opportunities program. Through this program NASA offers the use of many things: high-altitude balloons, landing platforms, and even parabolic aircraft flights. This program assesses proposals from various sources such as the government, academies, and even industries. Through this program, proposals that meet NASA requirements will be allowed to partake in the Flight Opportunities program. It is difficult to gain access to the program, yet this is one method of being able to perform microgravity experiments. One would have to submit a proposal from NASA and have this proposal accepted. They will then allow the group who had submitted to proposal to use their available environments for their experiments. This would be a very difficult and hard task to even be able to use NASA parabolic aircraft flights for zero gravity experiments.



Even though NASA does offer this program to be able use their parabolic aircraft flights for zero gravity experiments, it still doesn't make the testing readily accessible to use. After all, it requires a group to have their proposal be accepted by NASA just to use the parabolic aircraft flights. Even then the zero-gravity environment only lasts for 22 seconds. To be able to have multiple tests run on this aircraft would take a lot of time and would be difficult to get approved. For the flight to be able to simulate the zero-gravity environment it would constantly have to alter its trajectory. The aircraft wouldn't be able to do such a thing over small periods of time. Thus, resulting in the loss of time from one test to another, meaning multiple tests cannot be done over a short period of time.

While other alternatives do exist, it is very hard to actually make use of such alternatives. The availability of a parabolic flight is very hard to obtain. Even then multiple tests cannot be done shortly as it relies on the use of the flying of an aircraft which takes time. So, by creating a balloon that will instead simulate this environment would make zero gravity experiments more readily available. The goal of this project is to make the capsule simulate microgravity, for experimental tests to run. So, the results can be having the ability to have a readily accessible microgravity environment to run experiments.

Project Requirements

Mechanical Specifications

- Payload Mass: 5 to 10kg
- Payload Volume:
 - Diameter: 25 to 50cm
 - Height: 25 to 100cm
- Drop Time: 9 to 25sec
- Drop Height: 100,000ft

Ground Station Specifications

- Submit commands to balloon platform
- Store and display data from capsule sensors:
 - 3x Accelerometers
 - Temperature
 - Pressure
 - Humidity
- Display any errors transmitted from the capsule.

Design Considerations

Sensors

There will be several sensors inside the capsule measuring testing conditions and performance of the capsule. Almost all of these sensors will have their data transmitted to the ground station for display.

- Thermocouple (<https://www.adafruit.com/product/3245>) and amplifier (<https://www.adafruit.com/product/269>) (Appendix B, Datasheet 1)
 - Thermocouple can withstand and be read from -200°C to 500°C
- Humidity Sensor (<https://store.ti.com/HDC1080DMBT.aspx>) (Appendix B, Datasheet 2)
 - Relative Humidity Accuracy $\pm 2\%$
 - Can read temperatures with $\pm 0.2^\circ\text{C}$ accuracy, could be used for sanity checks of thermocouple or measuring operating temperatures inside the capsule
- Pressure Sensor (<https://www.arrow.com/en/products/ms561101ba03-50/te-connectivity>) (Appendix B, Datasheet 3)
 - Operating range: 10 to 1200 mbar, -40°C to $+85^\circ\text{C}$
 - Low Power
- 3x Triple-Axis Accelerometer (<https://www.adafruit.com/product/2019>) (Appendix B, Datasheet 4)
 - Configurable usage range from $\pm 2g$ to $\pm 8g$
 - 16384 possible measurement increments

Communications

To be able to communicate back and forth between the capsule and platform, the mechanical design team chose to use nRF24L01+'s (https://www.amazon.com/gp/product/B01IK78PQA/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1) (Appendix B, Datasheet 5). To simplify the design additions, they would need to make to communicate with our module, we chose to implement our design with the same transceiver.

We did consider using some other forms of data transmission but ultimately decided to use the nRF24L01+ due to the simplicity of required changes to the design and ease of transition to larger RF transceivers for full scale operation. In addition, the Mechanical Engineering team was already using this transceiver.



Figure 2 NRF24L01+ (hackspark.fr)

Other considerations for communications:

Module	Description	Link	Pros	Cons
nRF24L01+	2.4GHz RF transceiver	https://www.amazon.com/gp/product/B01IK78PQA/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1	<ul style="list-style-type: none"> • Cheap • Easier to replace with higher power RF devices • Already in use by the mechanical design team 	<ul style="list-style-type: none"> • Difficulty to work with at a higher level (requires low level setup and management)
XB24-AWI-001	Xbee Series 1 2.4GHz	https://www.mouser.com/ProductDetail/Digi-International/XB24-AWI-001/?qs=YPg7lQ8MWSexlGzr76XVYq%3D%3D&gclid=CjwKCAiArrrQBRBbEiwAH_6sNKPnD8RrSX6XOFu_51fuY_CzXio1TZd6p_pbyeXmO0HNU2JnDu33QLRoCdnAQAvD_BwE	<ul style="list-style-type: none"> • Easy to setup • Large community usage for support • Antenna can be upgrade on the module without rebuying 	<ul style="list-style-type: none"> • More expensive than the nRF24L01+ • Would be new system to implement on mechanical team's design
Alfa 802.11g/n adapter	1W high gain USB wifi adapter	https://www.amazon.com/gp/product/B0035GWTKK	<ul style="list-style-type: none"> • Can use popular libraries at high level • Easy to setup devices in local network mode 	<ul style="list-style-type: none"> • Completely different system needs to be implemented to use long range RF

Figure 3 Communications Considerations

Hardware

Sensor Interface

The sensor interface will be run using a Raspberry Pi that is connected to the internet (or locally to the database server) in order to be able to send sensor data to the database. Initially, it was suggested to use an Arduino to interpret the sensor data from the nRF2401+ but the extra overhead of interfacing with the Arduino seems unnecessary as reading the data from the transceiver is not a substantial usage of the total processing power of the Raspberry Pi.

We chose to use a Raspberry Pi because of the relative price compared to a more powerful computer system as well as the large number of libraries (such as wiringpi and py-spidev) and general community support for using the Raspberry Pi in some microcontroller-like capacity. The large number of programmable pins is also a large positive when prototyping especially with sensors or transmitters.

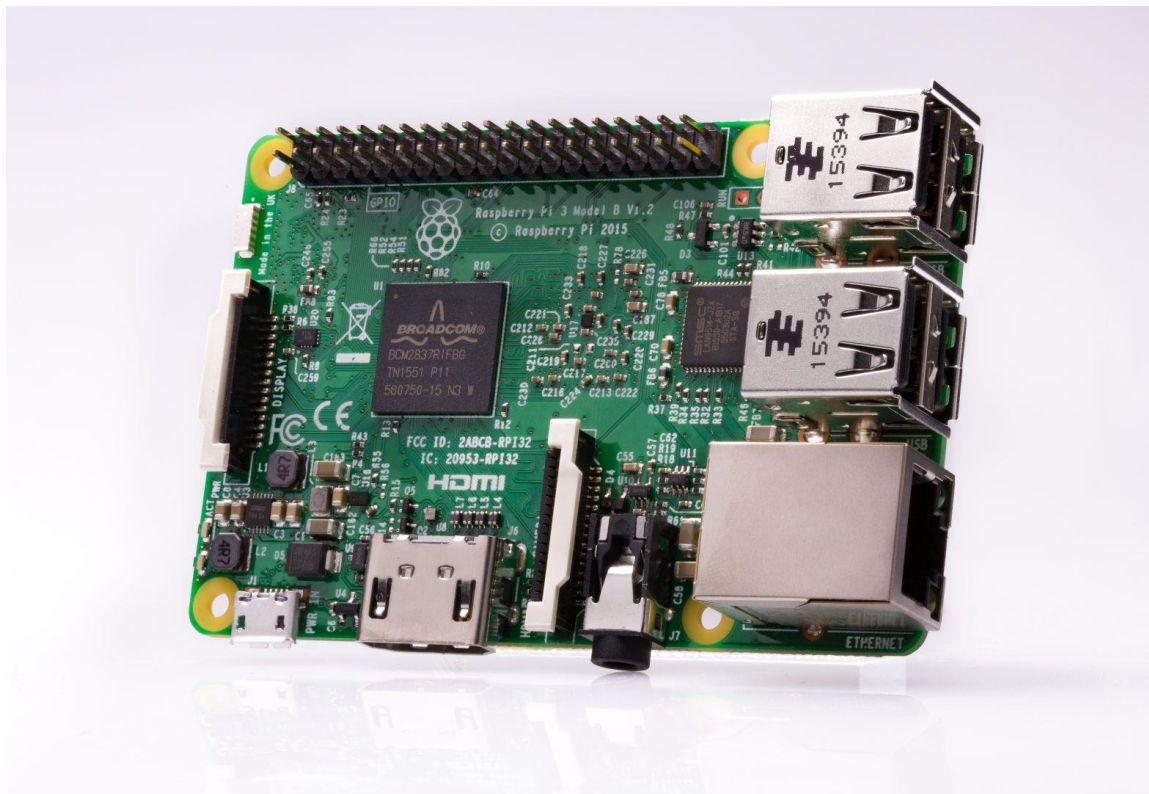


Figure 4 Raspberry Pi 3 Model B (raspberrypi.org)

The Raspberry Pi 3's that we will be using have the following specification based on the official specifications sheet provided by raspberrypi.org (<https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>):

- Quad core 64-bit Broadcom BCM2837
- 1.2 GHz clock speed
- BCM 43438 wireless LAN and Bluetooth Low Energy
- 40-pin extended GPIO
- 4 USB 2.0 ports
- 4 Pole stereo output
- Composite video port
- Full size HDMI port
- CSI camera port
- DSI display port which supports Raspberry Pi touchscreens
- Micro SD port for operating system and data storage
- Micro USB power input up to 2.5A

Database / Web Server

The web server is a virtualized Ubuntu 16.04 Server Edition. It is running on a Dell R610 at Andrew's house. The server itself has 48GB of EEC ram and two quad core processors. The virtual machine and its data is stored in a RAID configuration on a secondary data partition from the main functions of the server.

It made sense to go with this option because Andrew already had the server up and running. The primary operating system is Windows Server 2016 with Hyper-V so creating another virtual machine is relatively simple and won't require and dedicated hardware to be bought or acquired.



Figure 5 Dell R610 (dell.com)

The specifications for the server are as follows:

- 2x Intel Xeon X5550 quad core 2.66Ghz clock speed 8MB cache
- 48GB (4x 4GB DIMMs and 4x 8GB DIMMs)
- 2x 500Gb 7200RPM hard drives in RAID 1 for primary operating systems and other essential data (this drive array is backed up on a schedule)
- 4x 2Tb 5400RPM SSHDs in a drive pool with parity (see below section on that setup)
- 4x Gigabit ethernet connections

Storage Setup on Server

The two 500Gb drives are linked in a hardware-based RAID 1 using a Dell Perc 6/i RAID controller. This means that the drives are mirrored bit-for-bit. Having this arrangement double the read and write performance while providing 1 layer of security against disk failures (ie if one drive fails, all the data on the other is still available). This drive array is backed up to a cloud provider weekly on a schedule to provide an extra layer of security.

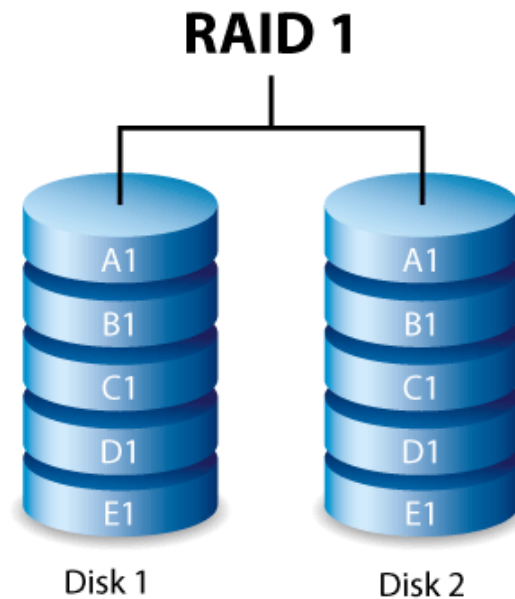


Figure 6 Raid 1 (seagate.com)

The 4 2Tb drives are setup as normal volumes to the primary operating system. The server is using an application called SnapRAID (<http://www.snapraid.it/>) which is a backup solution for multiple disks. It is an open-source program licensed under GPLv3. It works by assigning one of the drives you specify to be a parity disk for all the files on the other disks. This differs from a standard RAID 5 or 6 setup because the files themselves are not split among disks but remain whole on one of the disks. Some other key differences between SnapRAID and traditional RAID are:

- Disks do not need to be formatted and existing data can still be accessed
- Disks can be different sizes
- Disks can be added at any time
- There is no data lock-in
- If the number of failed disks is too many, the only data that is lost is the data on the failed drives

The SnapRAID setup allows for the disk redundancy benefits of RAID but the system still only sees the individual disks which makes creating data stores difficult. This is where another piece of ingenious software come in. DrivePool by StableBit (<https://stablebit.com/DrivePool>) is an application that takes multiple drives and combines them into a simple virtual volume that acts like a regular disk. It has features that include:

- Drive/Folder balancing - distributes files among the drives as they are being written
- Email notifications if any of the drives fall out of the pool (fail or are disconnected)
- Hierarchical pooling - Create virtual pools inside of other pools
- Interfaces for interacting with their other applications like mapping drives for cloud storage accounts or disk scanning and health reports

The downside to DrivePool is that there is a license fee and it is closed source. At the time of writing, a license for just the DrivePool application is \$30.



Figure 7 DrivePool logo (stablebit.com)

Software

Sensor Interface

To interface between the two Raspberry Pi's, we will be utilizing a library made for this type of transceiver, the NRF24I01+, called lib_nrf24 (https://github.com/BLavery/lib_nrf24). This is a python library that is based on the C library called NRF24 (<http://tmrh20.github.io/RF24/index.html>). Essentially, it takes care of the low-level management of the modules and provides a simpler python interface. It was written in 2014 and the author doesn't seem to be updating the library but it should work for what we need it to do. There are some examples in the repository specifically for the Raspberry Pi.

To get the transmitted sensor data into the database, we will be creating an API that will run on the Node application. The Raspberry Pi will need to send a POST command to the web server and it will add items to the database. This allows us to have the Raspberry Pi anywhere and the ground control application can be viewed anywhere there is an internet connection.

Database Software

As the majority of the data in this project will be handled by a database, deciding on the most suitable one is an important step. General things to consider when evaluating the appropriateness of a particular database are ease of learning, compatibility with other applications in use in this project, and general efficiency in handling mass loads of data.

For this project we will need to use a database that can easily store data received from the sensors aboard the balloon. Before any sort of decision is made on which database software to use, it is necessary to research the software beforehand. For this project there is a list of requirements necessary for the database software.

This list is as follows:

- The software must be easy to learn / easy to use
 - As the project has to be done relatively fast, a lot of time can't be spend using learning an entirely new software. Thus, the software to be used cannot be too complex to not learn relatively fast. So, having an easy to learn software will be very beneficial to us.
- The software must be able to store data given to it by a Raspberry Pi
 - The balloon device built contains a Raspberry Pi. This Raspberry Pi is how the sensors on said balloon will be sent to the ground system. As the ground system, and more importantly the database system must be able to receive this data as well as store it into the database.
- The database software must be able to run on a web server
 - In the current plan of the database system, the database shall actually run on the web server. It is here that the database will be able to receive the sensor information from the Raspberry Pi. Thus, it needs to be able to run on the web server for the current plan.
- The database software must be able to handle bursts of data

- The project is to run a great number of experimental tests. As to be able to keep track and store all the information of these tests, the database must be able to handle this large number of data. So, the software must be built for handling excessive amounts of data, in case that many experiments are performed.
- The database software must be relatively cheap / cost effective to be in the price range of this project
 - The pricings for database software is always something to keep in mind. We do have a max limit to how much we can spend, so checking the pricing for the use of servers (if needed) as well as licenses to use the software is something we will have to check on to make sure we don't go over budget.

MySQL



Perhaps the most well-known database software, MySQL is an industry standard. It is used in many professional applications and offers advanced security features among other standard ones.

Because it is an industry standard, learning and utilizing MySQL would be beneficial for future projects. However, while MySQL is easy to learn and use, other users have noted that it is slow when dealing with large tables.

MySQL is one of the most commonly used database management systems used in today's society. So, for the creation of this project's database MySQL was immediately the first thing that the group thought of to use. MySQL is a free open-source database management system. As it is used widely throughout many industries, those who can use MySQL are numerous. This results in a large community that are a part of MySQL. Thanks to this large community it should be easy to learn MySQL, as there would be many tutorials and such to help build a database using MySQL.

MySQL as relational database is built to be essential to open source PHP applications. Generally speaking PHP and MySQL database are used together for common web servers. In order to use MySQL, it must be used through the MySQL Workbench program. This program allows the users to administer and design their database structures. There are other interfaces that can work with MySQL, but MySQL has two different versions that make it more readily available.

Both MySQL and MySQL Workbench have a Community Edition that is free and open-source which can be downloaded directly from the MySQL website. MySQL also has a total of three other editions, Standard, Enterprise, and Cluster Carrier

Grade. Starting at Annual Subscription of \$2,000 up to \$10,000 for the more expensive editions of MySQL. There is a huge markup of the price of these other editions, which are clearly result in being out of the price range for this project. If MySQL were to be used the Community Edition version which is free and open-source shall be used.

Upon further research it seems as if MySQL works very well as a database system for the average case. The implementation of MySQL seems to work best for small to medium web pages. Meaning MySQL will work better when not having to deal with extremely large tables of data. In this project it is assumed, that the data gathered from the experiments shall be large in number. This may negatively impact the use of the database if the many experiments were conducted. As it would increase the amount of data stored, and negatively impact the performance of MySQL.

Through the use of MySQL workbench, it has been confirmed that through the creation of a server, one can connect a Raspberry Pi to MySQL. MySQL database would merely need to be configured to accept these external connections. After having configured MySQL, and editing the Raspberry Pi, data from the Pi can be sent to MySQL. As we are using our own Raspberry Pi in this project, it is very useful to know that it is possible to do so. There also seems to be a lot of tutorials online and information regarding how to set up the Raspberry Pi to connect to MySQL. Thus, it shouldn't be too hard to perform based off of all the online examples.

Below is a figure that demonstrates a simple entity table that would be used in MySQL Workbench. The MySQL Workbench is what would be used when writing and designing the database for our system. Below is merely an example of what the results would seem to be. Of course, the figure doesn't show much, but it does demonstrate that using the MySQL Workbench isn't that difficult to comprehend. So, it wouldn't be too hard to actually create the database using MySQL.

MariaDB



Another widely-used database software, MariaDB was made by the same developers as MySQL, and thus is an alternative that has the same core functionality. Also, like MySQL, it prioritizes security and has several accompanying features as well.

While MariaDB can handle lots of site traffic and simplify things such as integration between SQL and NoSQL, it can still be on the slow end when actually updating tables.

MariaDB is another relational database management system similar to that of MySQL. In fact, MariaDB is made by the same developers who originally made MySQL. Due to being made by the original creators of MySQL, it maintains many similarities to MySQL. Having high compatibility with MySQL, as well as even having the same APIs and commands that MySQL has as well.

Similarly, to MySQL, it is also free and open source. In fact, some of the features that are in the paid versions of MySQL are in the free open source of MariaDB. As it is open source, everything about it including bugs, development plans are made public. As open source it is supported and developed by the community surrounding it.

As stated MariaDB is extremely compatible with MySQL. But it is generally believed that MariaDB is more efficient than MySQL. The MariaDB has better performance when compared to MySQL on complex queries. This is due to the engine that MariaDB is running on, the Aria engine. MySQL runs on InnoDB engine, and due to these two different engines results in the fact the MariaDB generally has better performance.

While being very similar to that of MySQL it is possible to run MariaDB with the Raspberry Pi. By performing similar the similar acts, one would do to connect

MySQL to a Raspberry Pi, the same can be said for MariaDB. On the current newest version of the Raspbian, the MariaDB should already be present in the repositories. Meaning all one would have to do is install MariaDB on the Raspberry Pi and the connection from the Pi to the database should be simple. It also seems there are a lot of tutorials to help one try and make the connections from MariaDB to the Raspberry Pi, so it wouldn't be too hard to accomplish this task.



Amazon Aurora is another relational database. Unlike the others this was built for the cloud. That the database would be implemented onto the cloud, where everything is stored. It is also compatible with MySQL, and PostgreSQL. It was made to have high performance and availability, as well as being simple and cost effective as an open source database.

Amazon Aurora is managed by Amazon Relational Database Service. The Amazon Relational Database Service does many of the administration tasks that the user would normally have to do in another relational database software. Aurora is said to be up to five times faster than MySQL, and three times faster than PostgreSQL. This would mean that using Aurora would be much more beneficial than that of MySQL. As it would work faster than it and do a good job

One of the things Aurora does is that it will automatically grows storage for the database you use. For each database instance it has a maximum capacity of 64TB. As the project does create more data over the course of each experiment, having the ability to have the database expand into more storage would be very beneficial. This would allow us to not have to worry about going over the capacity of how much storage we have available when using other database software, like MySQL.

The problem with Aurora that MariaDB and MySQL do not have is the pricing. Aurora is open source like the two but does not provide a free version. Instead the pricing for Amazon Aurora is determined on how much storage you use. In the region of US East on the Amazon website for purchasing Aurora, it lists the

going prices. The Storage Rate is \$0.10 per GB every month, while the I/O Rate is \$0.20 per 1 million requests. This essentially saying that for every GB of data we use on the cloud without database, we would have to pay \$0.10 every month. While I/O rate dictates how many times we can send requests to the database on the cloud. I/O rate doesn't have monthly rate, but instead goes by the number of requests. If the number of requests were to exceed 1 million we'd be charged an extra \$0.20. Of course, these prices aren't incredibly expensive it is still something to keep in mind. But it is believed that the amount needed to pay on storage as well as the number of requests needed to make would not go out of the price range. After all you only pay for the storage IO requests that are made. So, it is possible to use Amazon Aurora as a database on the cloud for this project.

As Amazon Aurora is on the cloud, the idea of using a web server and having the database hosted on that would be pointless. Instead if Amazon Aurora is used we would instead have the web server make the requests from the database on the cloud. The other problem that would be looked into is how to connect the Raspberry Pi to the database on the cloud. It seems as if there is a way described on the Amazon website to connect the Raspberry Pi to AWS. The AWS stands for the Amazon Web Service, which is the Amazon's cloud service. So, by being to connect to AWS, we would then be able to connect to our specific database for this project.



MongoDB is a free and open source database program. It is a document-oriented database software, where the documents are JSON-like. This is the leading NoSQL database software. MongoDB even has cross-platform compatibility. MongoDB allows for fast and flexible access to the data stored in the database. One of MongoDB's major points is how scalable it actually is.

Unlike the database software previously mentioned, MongoDB stores its data into flexible JSON-like documents. This allows for the fields of these documents to vary from one document to another. It even allows the data structure to be changed over time as well. These documents allow for the storage of large volumes of data. As this project includes the storing of data from many experiments, the results could give out large volumes of data as well. Thus, using MongoDB in tandem for this project may be extremely useful.

MongoDB is also compatible with .NET applications as well as the Java platform. Not only that but MongoDB is often used with Node.js, which is mainly used for building server-side applications. If on the front-end of the ground system Node.js was being used, then it would be easy to try and implement MongoDB as the database software.

Upon closer research a tutorial was found to install MongoDB and Node.js on a Raspberry Pi. As mentioned, this project uses a Raspberry Pi to receive information data from the sensors aboard the balloon. This tutorial includes the steps to install both Node.js and MongoDB onto the Raspberry Pi. Beyond that it even includes the methods of what to do to have the Raspberry Pi interact with the web server as a service. If taking this route of using the Node.js along with MongoDB this tutorial would be quite helpful to the project in getting the connections to everything started.

As the MongoDB stores it as JSON-like documents it is important to go over what JSON actually is. JSON stands for JavaScript Object Notation. This is a format of

having data be structured in an organized way for it to be readable. JSON's primary purpose though is to usually transfer data between a web server and a web application. As that is part of the current design of this project it would be useful to know more about it. Generally, for machines it isn't difficult to parse and generate JSON data which is why they are very good for transferring data. JSON has two structures, objects and arrays. The way it is presented is so that each name of a key has its own value. An object is merely one instance of some sort of variable having a value, while an array is a list of these variables that have a value.

In the below figure from the MongoDB website, is an example of what the JSON-like documentation of the data should look like. This specific example, has embedded data, where the phone and email are listed under the contact. In this example though, are both instances of an object and an array. The object is a single instance, which is the line `username: "123xyz"`. The key is `username`, while the value of the username is `123xyz`. While the array is `contact: {phone and email}`, for this example the array's key value is `contact`, but its data is a list of objects that represent the phone and email. As a general basis this is how MongoDB stores its own data.

```
{
  _id: <ObjectId>,
  username: "123xyz",
  contact: {
    phone: "123-456-7890",
    email: "xyz@example.com"
  },
  access: {
    level: 5,
    group: "dev"
  }
}
```



PostgreSQL is another relational database management system. It too is also a free and open source database system. Originally designed to run on UNIX platforms, it was redesigned to be able to run on other platforms like Windows and Mac OS. The source code for PostgreSQL is available under its own license. Due to its stability it requires very little maintained efforts, this results in a lower cost of ownership compared to other database management systems.

One of the functions of PostgreSQL, is the ability to add custom functions through programming languages. As useful as that is, it is currently unknown as to whether that would be necessary for the current project. Nevertheless, the ability to do so is something to be considered as to why PostgreSQL is useful as a database software. Another function, is that PostgreSQL allows its users to define their own data types, and index types.

Similarly, to the other database systems, PostgreSQL is also able to be installed on a Raspberry Pi. Just like the other database systems, it is possible to have the Raspberry Pi run an instance of a PostgreSQL database server. This perhaps is always the most important feature to look out for when researching the database software. After all the whole design of the project depends on the use of a Raspberry Pi to send data to the ground station. Since if any of the database software that were researched were unable receive information from a Raspberry Pi, we probably wouldn't dare use it.

One thing about PostgreSQL is that it can handle large amounts of data in its database. It can have a database storage be in the TBs of data and still work. Of course, this is dependent on the hardware that's being use to run the database server. But when the tables grow in the database to extremely large numbers it

can slow down queries of the database. So as long as we don't have enormous table sizes PostgreSQL should work fine.

Chosen Database Software

After careful consideration, and research done, the Database software that will be used for this project is MySQL. There are many reasons as to why it was chosen, which will be explained later. It would be best to start as to why the other options were not chosen.

MariaDB is a very useful database. Having been made by the same developers who created MySQL. It does seem to do a better job than MySQL, especially since it supposedly has the features of the paid versions of MySQL for free under MariaDB. As it is very similar in syntax to MySQL, it wouldn't be too hard to learn how to use MariaDB. Unfortunately, MariaDB doesn't have its own management tool, like how MySQL has MySQL Workbench. So instead for MariaDB one would have to use the Database Workbench instead (which still functions primarily the same the MySQL counterpart). So, we would have to go through more work in using it than we would have to for MySQL. Also, MariaDB supposedly runs slowly on updating tables which is necessary for our design.

Unlike MariaDB and MySQL, Amazon Aurora was almost chosen as the database system software for this project early on in the development. This was because one of the previous designs was to instead have the database be on the cloud instead of on a server like it is now. It was thought to be much easier to build the design of the project when the database was on the cloud. As the only thing required was that the Raspberry Pi had to merely connect to the cloud and upload its data to the database. This was thought to be an easy way of obtaining our data. Of course, this didn't happen but it was put into careful consideration. Not only that, even if Amazon Aurora does require money to pay, unlike MySQL and MariaDB, it isn't such a large amount of money for the project's budget to go over on it. Especially since one of Amazon Aurora's main draw is that you pay for the data you use. Nevertheless, even if we aren't going to be using Amazon Aurora it will still be there as a backup. In case MySQL didn't work, we would most likely fall back onto using Amazon Aurora for the projects database system software. One downside of using Amazon Aurora is that there is a latency between the application or client and the database server.

PostgreSQL is a very different database system from the others. It has the ability to add custom functions through programming languages. It does focus more on its extensibility, allowing the user to define its own data types and index types. These functions sure are very unique, but don't seem to be really necessary in this project. This portion of the project is mainly about storing data from the sensors on the balloon. These sensors already dictate what data types we will be collecting. Thus, it is unnecessary to have the ability to define our own data types for this database. So, the uniqueness of PostgreSQL is merely there for those

who need it, which this project doesn't. Thus, it isn't really necessary to have PostgreSQL be the database software this project uses.

MongoDB was going to be the database we used in our system originally before MySQL. We had found out about the MEAN stack. The MEAN stack is a free and open-source JavaScript software bundle, which includes, MongoDB, Express.js, AngularJS, and Node.js. After we had done some research on Node.js and such we found out that it the MEAN stack would fairly well with our project. But the only problem was whether or not we wanted to use MongoDB as our database. As MongoDB is a non-relational database. Unfortunately, all of our data that we had planned was to be stored in a relational database. At first, we were not sure whether or not MySQL could be used in the MEAN stack and how well it could work. So, all early work of this project was decided to be done in MongoDB. After discussing things, we had decided it'd be better for us to store our data in a relational database as opposed to a non-relational one like MongoDB.

The one thing about all of the databases that was researched was that they can all connect to the Raspberry Pi. Because of this, it seems as if any of the databases that were researched could be used for this project's design. Yet the database system software that was chosen was MySQL.

MySQL was chosen as our database system due to many reasons. One of these reasons was that upon our research it was discovered that one of the most popular database management systems is MySQL. We also wanted to make sure the other software we used would work well with MySQL. As it turns out Node.js (which we had already decided on using) and MongoDB work very well together, which is why we had planned on using MongoDB instead. But, after originally planning to use MongoDB we realized that the data we decided to store and formatted fitted a relational database better than a non-relational could. So, we had switched to using MySQL instead of MongoDB. The front-end application will be using Node.js as its basis. As it turns out Node.js and MongoDB work very well together, but fortunately we were able to find a way to replace MongoDB with MySQL. Thus, allowing us to use the MEAN stack which we had planned on using, without the inclusion of MongoDB but with MySQL instead. Even if we are able to use MYSQL as a replacement for MongoDB in the MEAN stack isn't the only reason why we chose it.

MySQL supports JSON-like document data as of version 5.7.8. As mentioned before (in the section of MongoDB) these are documents that simply contain the data that was collected. These documents are set up in a way to make them easily readable. The documents follow a set pattern and have a relatively easy way of being parsed and generated by machines. It is also easy for humans to

read and write these documents as well. The JSON documents are based off of two structures, that of an object and an array. Almost all modern programming languages will recognize and support these two universal data structures. Due to the simplicity in parsing and generating the JSON documents as well as the fact they are based off of two universal data structures in modern programming languages, it becomes very easy to transmit this data. Thus, being able to transfer this data from MySQL to Node.js becomes much easier since the data being transferred is in a very easily readable format. After all the front-end application of the project design will have to display the data, so having the data transfer from the database to the front end rather easily is something very helpful to the project as a whole.

The thing about MySQL is that it is a free and open source database system. Many of the databases that were researched were the same in fact. By using the MySQL database software, we'd be saving money. We wouldn't need to have to worry about going over our budget. We also wouldn't have to worry about how much data we are storing or how many requests to the database we have done, like we would if we had chosen to do Amazon Aurora as the database software. There is a paid version of MySQL out there that do include more options for us. Though everything that we need to do and accomplish can be easily done in the free version. Even if MySQL is free, we do still have to worry about how large the data we are storing will get.

We have decided ourselves that we would be the ones hosting the server for our MySQL database. Meaning as we are hosting the server ourselves we do not have to worry as much about much data we should allocate for storing data into MySQL, since we are the ones who can decide that. We also won't have to worry about MySQL being given multiple operations at a given time. At most we are mainly seeking data that is related to singular test that was performed over time. So, we won't have to worry about poor performance issues at all. MySQL also has the MySQL Workbench, which is what would be used when writing and designing the database for our system. After all, through this we can connect to our Raspberry Pi. This would make the whole idea of making our database simple to make and understand.

We have already found multiple tutorials on how to use MySQL along with the Raspberry Pi. It seemed as if every database system software had multiple tutorials on how to download and use the Raspberry Pi along with said database system software. Yet we found a great many tutorials describing how to use MySQL along with Node.js on a Raspberry Pi. From these tutorials we were able to figure out how we could connect a Raspberry Pi to our system. Unfortunately,

we were not able to send the data from the Balloon to the database directly, so instead a script was written that would simulate this transfer of data instead.

Due to all of these reasons is why MySQL shall be used as the database software for this project. That the data we are collecting from these experiments is can be easily stored in a relational database like MySQL. As we can easily implement it alongside Node.js. Have easily transferable files between the database and front-end application. MySQL is a free and open source software to use, thus we don't need to spend any expenditures on purchasing any unnecessary database software. The ability to sync MySQL and a Raspberry Pi up via a server on MySQL Workbench, is necessary for our project. The last bit of why we should use MySQL, is how easy it is for the team to learn how to deal with it. As there are many tutorials and guides out there on we should use MySQL due its overwhelming popularity in the industry. Because of all of this knowledge available to the team, using MySQL as the database software for this project should be extremely beneficial.

Web Server/Application

Server Frameworks

As the application will need to handle communication between the device sending data and the database storing it, having a proper server framework will facilitate the process and make the entire application more efficient.

Node.js + Express

While not technically a web framework by itself, Node.js is a very popular server in web development. Unlike most other servers, it does not deal with PHP or Java, and instead simply runs JavaScript server-side. It provides asynchronous, event-driven I/O, and greatly assists in handling requests.

Because it uses JavaScript, it is easy to learn, and the same code is used on both the server and client sides. However, other users have noted that Node.js lacks proper error handling and memory management, and it is also difficult to maintain the code.

Express.js is specifically a web application framework built for Node.js. It is very minimalistic, but there are many features that can be added depending on what is needed. One thing to note is that Node.js + Express is ideal for highly-interactive single-page applications.



Django

Django is a widely known web framework for Python, and it includes many functionalities and features out of the box. It is well-known for its high degree of

scalability, customizability, and security when creating and managing projects, and are thus its most notable features.

For a time after its inception, Django was very fast when executing statements, but that has faded with time. Because the developers have focused on backwards-compatibility, its speed has suffered by a steadily increasing amount as a result. Its suite of included features can also be a disadvantage, as every project likely won't need every single functionality it comes with. Additionally, because Django is a WSGI (Web Server Gateway Interface)-based server, its protocol is synchronous and thus not suited for developing dynamic, real-time applications.



Web Application Frameworks

<https://thenextweb.com/dd/2015/04/21/the-14-best-data-visualization-tools/>

While not strictly necessary for the development of a webpage or web application, having appropriate frameworks can greatly simplify the creation. Development that would normally require hours of coding and research can be cut down significantly on time by not being restricted to basic HTML, CSS, and JavaScript.

AngularJS

Because standard HTML is not known for being very dynamic, AngularJS addresses this problem by extending HTML vocabulary. With two-way data binding and DOM manipulation, the result is a highly dynamic web page for the user.

The main complaint for AngularJS is that it can be notoriously difficult to learn if one is unfamiliar with its features or with MVC architecture in general, but when one considers its speed and responsiveness in the end result, a learning curve may be an acceptable flaw.



Bootstrap

While used primarily for designing mobile-friendly websites, Bootstrap is still a popular framework for any kind of web development. It is an HTML, CSS, and JavaScript framework that is widely used to build responsive web applications, but many people use it for its stylesheets and jQuery plugins as well.

Although Bootstrap is very useful for easily developing responsive websites, it does come with restrictions. Because of how much it simplifies the website

design process, a lot of code rewriting may be required if one wants to deviate from the standard design put forth by Bootstrap. Thus, it is worth verifying beforehand if Bootstrap's default designs are similar to what is desired before putting time into designing with it.



Data Tools

The purpose of this project is to create meaningful data from experiments for researchers to study, and the easiest way to spot correlations between sets of data is to look at the data in the form of a chart. While a chart could easily be created from the data after the conclusion of the experiment, there is much more merit in dynamically creating graphs for researchers to observe in real-time. Thus, looking at different data tools for dynamically creating graphs is a necessity.

D3.js

Data-Driven Documents - better known as D3.js - is one of the most popular data visualization tools out there. Using HTML, CSS, and SVG, it creates spectacular graphs in modern browsers. D3 offers nearly every kind of chart and diagram in existence, and it is also highly interactive and filled with many features.

D3 leaves nothing to be desired in terms of what is offered, but there are still a few considerations. It will only work in modern browsers - which may not necessarily be feasible depending on the project - and it also has a notoriously steep learning curve.



Chart.js

Chart.js is a simple tool for creating dynamic charts using JavaScript. It can create several different types of charts from multiple sets of data at once, and it features smooth transitions when dynamically updating data in the chart.

Chart.js does not have many features, but the features it does have it executes spectacularly. It is fast and responsive, so it is ideal for applications that only need to draw simple charts.



Chart.js

Google Charts

Google Charts is a web service for creating charts from user-supplied information about data and formatting. It offers many different types of charts, and all are capable of dynamically updating themselves with changing data. There are also a fair amount of customization tools available for use, so making an ideal chart for a set of data is relatively simple.

It is worth noting, however, that while Google Charts does have some customization options, it does not offer as many as some other packages do. In addition, usage of Google Charts requires an internet connection at all times, so it may not be ideal for use in some applications.



The MEAN Stack

Special mention must be given to the MEAN stack; it is a software stack consisting of MongoDB, Express.js, AngularJS, and Node.js. This stack uses Node.js + Express on the backend for connecting the MongoDB database to AngularJS on the frontend. Everything in this stack essentially utilizes JavaScript exclusively, so there isn't a need to try to incorporate many different languages in one project. There are some restrictions inherent in using only one language, but this software stack neatly ties multiple different components of the same project together.



Implementation Details

Sensor Interface and Communications

The following steps will show how to configure the Raspberry Pi's to communicate with one another and how to get the receiving Raspberry Pi to talk to the database. Any lines starting with \$ and in red font are commands or text that is meant to be entered at the command line.

These steps assume that the Raspberry Pis have Raspbian Lite (<https://www.raspberrypi.org/downloads/raspbian/>) installed. The Raspberry Pi's that were used to write this section have Raspbian Stretch Lite kernel version 4.9 installed. It is recommended to create another sudo user account and disable the default pi login for security.

1. First, we need to make sure that SPI is enabled as it is the way the Raspberry Pi communicates with the NRF24L01+.
 - a. This setting can be accessed from Raspberry Pi config that is run with the following command
`$ sudo raspi-config`
 - b. Choose option 5 - Interfacing Options
 - c. Choose option P4 - SPI
 - d. Select Yes to enable the SPI interface
 - e. Select Finish to exit the configuration utility
2. Now that SPI is enabled, we need to install the python modules
 - a. The first module is the python-dev (<https://packages.debian.org/stretch/python-dev>) which contains development tools for python that include embedding python and extending the python interpreter
`$ sudo apt-get install python-dev`
 - b. The next module is python-rpi.gpio (<https://pypi.python.org/pypi/RPi.GPIO>) . This module extends the Raspberry Pi's GPIO header to be used by python applications. It is

an essential part of being able to use python to access the transmitter.

```
$ sudo apt-get install python-rpi.gpio
```

- c. The last module we need is the lib_nrf24
(https://github.com/BLavery/lib_nrf24) which is a python port of the NRF24 C library
(<http://www.airspayce.com/mikem/arduino/NRF24/>). This module takes care of all the low-level management involved in using the NRF24L01+ module and provides an interface in python to make development faster and easier.
 - i. Unlike the other modules, we have to clone the GitHub repository. This requires git but it should be installed by default on Raspbian. First, we will create a folder to store the module in. For this project, we installed the module in ~/lib_nrf24 (~ denotes the user's home directory).

```
$ cd ~  
$ mkdir lib_nrf24
```
 - ii. Now we will clone the repository.

```
$ git clone https://github.com/BLavery/lib_nrf24 lib_nrf24
```
3. Now we can go ahead and wire the modules to the GPIO pins on the Raspberry Pi. For this project, we wired them according to Figure 22.

Module Pin Name	Raspberry Pi Pin Name	Raspberry Pi Pin
V+	3v3	17

GND	Ground	25
CSN	GPIO_CE0_N GPIO 8	24
CE	GPIO 17	11
MOSI	GPIO_MOSI GPIO 10	19
SCK	GPIO_SCLK GPIO 11	23
IRQ	Not connected	Not connected
MISO	GPIO_MISO GPIO 9	21

Figure 22 NRF24L01+ to Raspberry Pi pin connections

4. Now we can test the setup of the two modules and the Raspberry Pi's
 - a. Navigate to the folder in which the lib_nrf24 is installed
`$ cd ~/lib_nrf24`
 - b. On one of the Raspberry Pis, start the example receiver and on the other start the example sender. These have to be started as root because they are accessing the GPIO pins.
`$ sudo python example-nrf24-recv-rpi.py`
`$ sudo python example-nrf24-recv-rpi.py`
 - c. If everything was configured properly, the sender should transmit "HELO" to the receiver.

Database

Database Diagrams

The following diagrams are what are known as an ERD. An ERD is an Entity Relationship Diagram, they display the relationship between different entities in a database. These entities are what hold and house the data that are stored in the database. Thus, an ERD shows how each set of data is grouped together, as well as how each group relates to one another. When creating a database, it is best to model how it shall store data. The following ERD's were made to try and emulate this.

When trying to create the database for this project, a total of 3 ERDs were made. In order of creation they go from ERD 1, ERD 2, and ERD 3. With ERD 3 being the planned model for the database. As a sense of progression ERD 1 functions as a precursor to ERD 2. While both ERD 1 and ERD 2 serve as a precursor for ERD 3. This results in having the ERDs have the same entity tables from one ERD to another. But while they do have similar entity tables, all three ERDS function differently. Because of these differences the each ERD shall be explained individually.

The specifics of the ERDs that were made follow a set of rules. The first being that at the top of each entity (the boxes) is the title of that entity. This designates the description of what that entity is about, and the information it holds. Below the title are a list of Primary Keys and Foreign Keys (if any), also referred to as PK and FK respectively. Below these keys, is a line that sperate the keys and the data. This is the data that is primarily stored in each entity. Beyond that are the lines between each of the entity tables. These lines describe the relationship between the entity tables that are connected. Everything shall be described more accurately in the descriptions for each ERD.

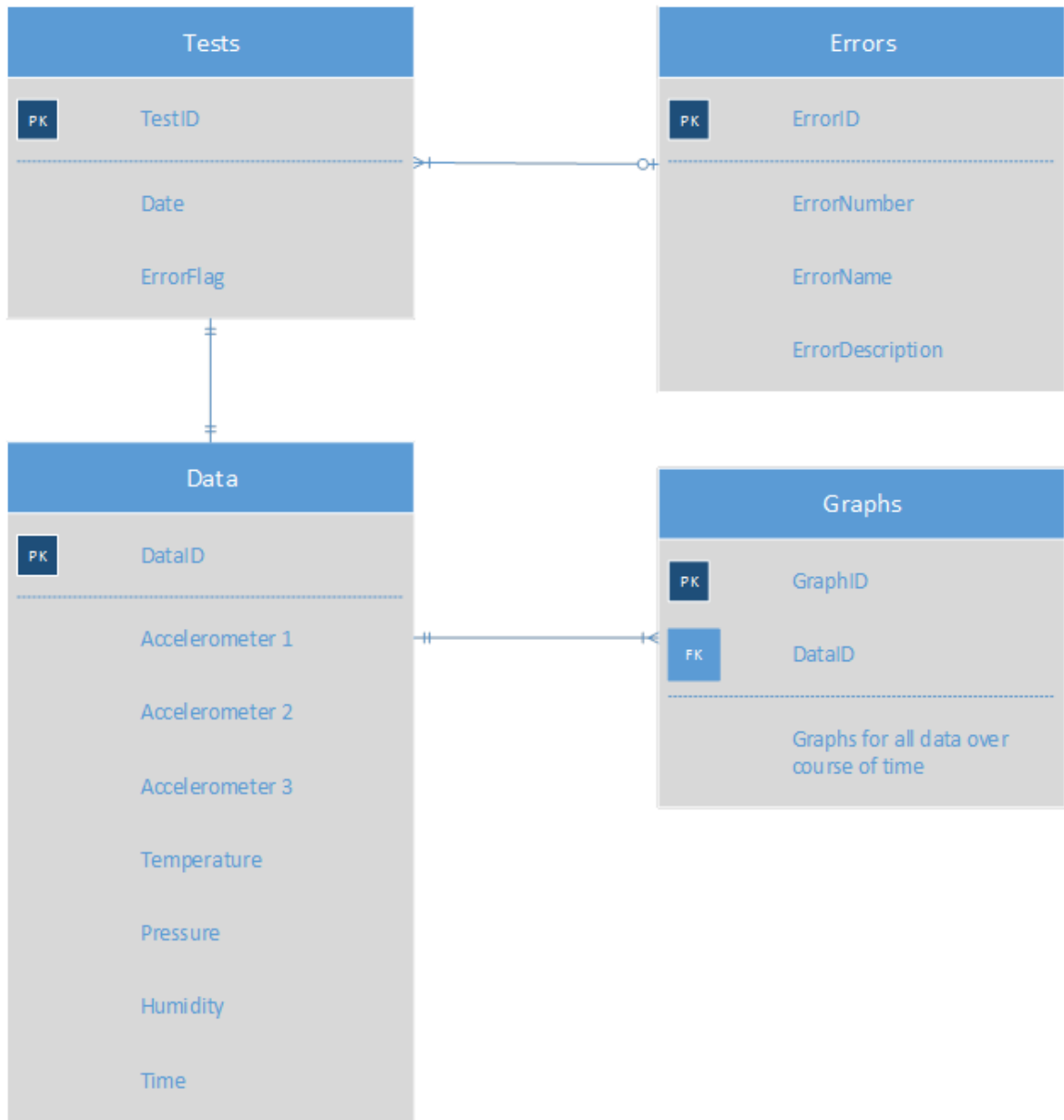


Figure X1 ERD 1

ERD 1 Explanation

The above diagram designated as ERD 1, is a rough draft of the database model. This was the first ERD that was made for the database to be modeled after. ERD 1 has a total of four entity tables: Tests, Data, Errors, and Graphs. Each of these tables has at least one relationship with each other, and a reason as to why they are made.

To start off the first and primary entity table is referred to as the Tests table. The Tests table shall keep track of very little, but important information. The Tests table will keep track of the Date of the experiment, as well as a variable known as ErrorFlag. The Tests table will also have its own Primary Key (PK in the ERD) known as TestID. The TestID is a unique number given to each experimental test undergone by the reduced gravity experiment balloon. When a new test is ran by the balloon, that test will always have a unique number to differentiate it from other tests. We will also keep track of the Date said experiment was performed on, as it may be important information to retain for each experimental test. Lastly, we will have the ErrorFlag value represent the success of the test. If the test is performed by the balloon without any errors the ErrorFlag will be set to false. But if some sort of error does occur, such as a sensor not work, the ErrorFlag be sent to true. If the ErrorFlag is false there are no errors; while the ErrorFlag is true some error had occurred during the experiment.

The next table is the Errors table. This table similarly to the other tables has its own Primary Key, ErrorID. This is used to identify each error differently from another error that was possible. We will also store the ErrorNumber which represents which error that had occurred, as well the same of that error under ErrorName. Beyond that each error will have its own ErrorDescription, to explain what that error is so the user can try and fix it. The one thing about this table in particular is that it will only be filled out if the ErrorFlag in the Tests table is set to true. Reason being is that if ErrorFlag is false, we don't have an error so there is no reason to establish anything inside this table. So, everything in Error table on an errorless case will be set to 0 and null, to indicate that no errors occurred.

The Data table works differently as it does in the other ERDs. In ERD 1 the Data table merely says that it will store various kinds of data from the experiment. It will like others have its own Primary Key, DataID, to differentiate the entity tables data from other sets of data. But it will also keep track of the various information of the multiple sensors. It will hold data from the three different accelerometers, temperature, pressure, humidity, and time of said experiment. The problem with

this data, is that the relationship it has with Tests table is that there will only be one set of Data collected. Meaning that we will only have one Data table that houses all the information. This is wrong, as we are collecting data over a course of time through the experiment. The experiment includes collecting data over 10 to 30 second period from the sensors. It was planned that maybe the data from the sensors would be averaged out over this time period and then placed into the Data table. While the time was to be the number in seconds of how long the experiment took.

While the Data table acts as an average collection of the data from the time period of the test, the Graph table was implemented to try and fix this. The Graph has its own Primary Key, known as GraphID. The Graph table was to have the Foreign Key (FK in the ERD) of the Data Table DataID, this was to help with the creation of the graphs. The Graph table was supposed to function as a collection of graphs. These graphs were to be the values of the different sensors, that were mentioned in the Data table, over the course of the experiment. Essentially each sensor was to have its own graph charted over the course of the experiment. The Foreign Key was implemented to connect that these sets of graphs were tied specifically to this set of Data.

There were many reasons as to why ERD 1 was decided to not be used as the model for the database. The main reasons were the implementation of the Data table and Graph table. These two entities contradict how we want to store the data from the experiments. As the experiment is done over a course of time, merely having one set of data points or even the average would not help much when one was to look at the results. So, having only one Data table wasn't going to help much in data observation. While the graphs are nice to have, it would make this project much harder/more complex than what we wish to do. Also, since the creations of both ERD 1 and ERD 2, the group has been advised that it might be better to not try to store graphs in a database. Thus, not having a Graphs table as well storing more than just one Data table will be a requirement for our database. Unfortunately, ERD 1 does not satisfy this and was not used as the model for our database.

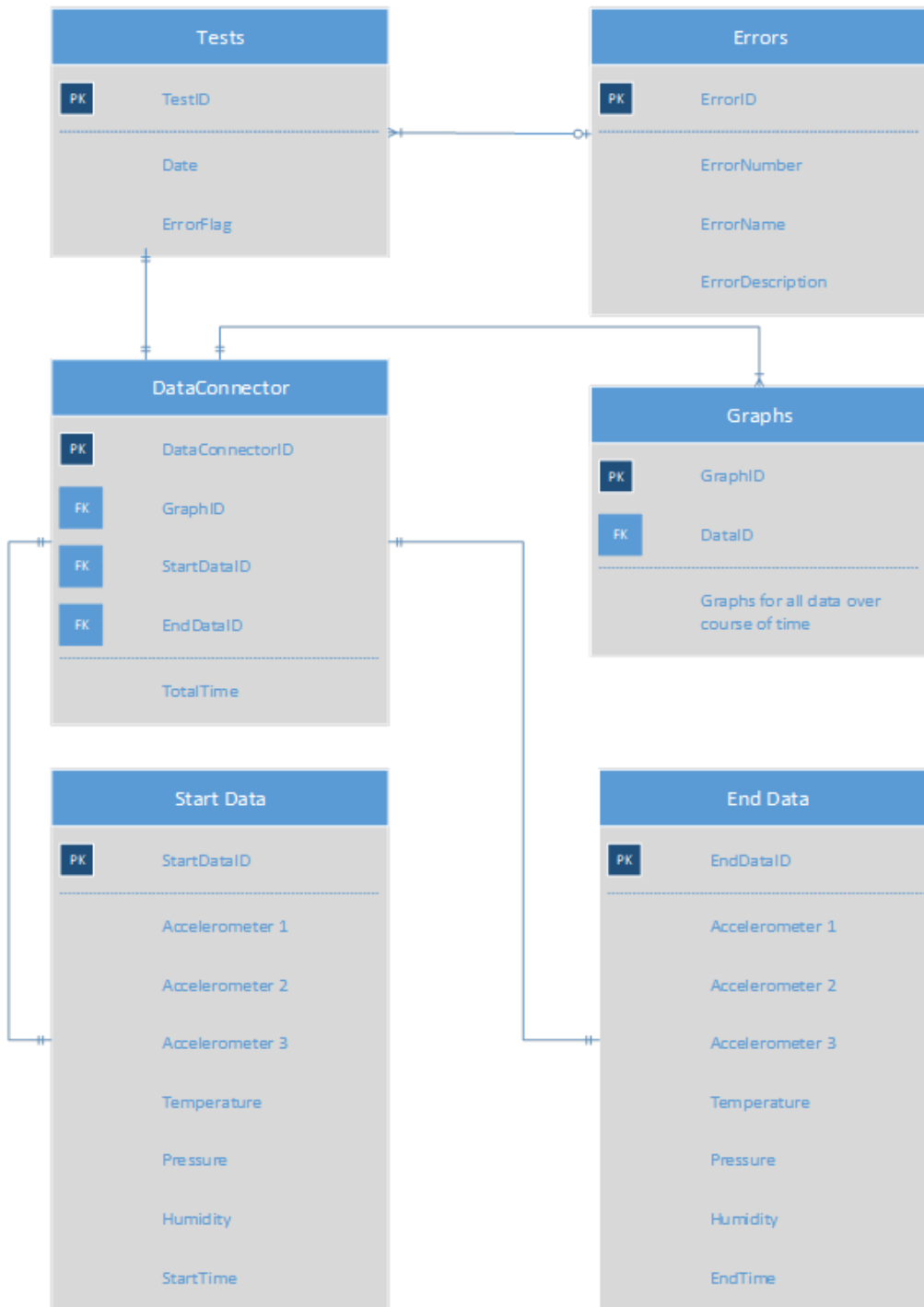


Figure X2 ERD 2

ERD 2 Explanation:

The above diagram designated as ERD 2, is another rough draft of the database model. This was the second ERD that was made for the database to be modeled after. ERD 2 has a total of six entity tables: Tests, Errors, DataConnector, StartData, EndData and Graphs. Each of these tables has at least one relationship with each other, and a reason as to why they are made.

Similarly, to ERD 1 the Tests table and Errors table perform in the exact same way. The following two paragraphs merely reiterate what was said about these two tables previously. The only difference between ERD 1 and ERD 2 is the data collection from the experiments would be handled.

To start off the first and primary entity table is referred to as the Tests table. The Tests table shall keep track of very little, but important information. The Tests table will keep track of the Date of the experiment, as well as a variable known as ErrorFlag. The Tests table will also have its own Primary Key (PK in the ERD) known as TestID. The TestID is a unique number given to each experimental test undergone by the reduced gravity experiment balloon. When a new test is ran by the balloon, that test will always have a unique number to differentiate it from other tests. We will also keep track of the Date said experiment was performed on, as it may be important information to retain for each experimental test. Lastly, we will have the ErrorFlag value represent the success of the test. If the test is performed by the balloon without any errors the ErrorFlag will be set to false. But if some sort of error does occur, such as a sensor not work, the ErrorFlag be sent to true. If the ErrorFlag is false there are no errors; while the ErrorFlag is true some error had occurred during the experiment.

The next table is the Errors table. This table similarly to the other tables has its own Primary Key, ErrorID. This is used to identify each error differently from another error that was possible. We will also store the ErrorNumber which represents which error that had occurred, as well the same of that error under ErrorName. Beyond that each error will have its own ErrorDescription, to explain what that error is so the user can try and fix it. The one thing about this table in particular is that it will only be filled out if the ErrorFlag in the Tests table is set to true. Reason being is that if ErrorFlag is false, we don't have an error so there is no reason to establish anything inside this table. So, everything in Error table on an errorless case will be set to 0 and null, to indicate that no errors occurred.

The real difference between ERD 1 and ERD 2 is shown in the DataConnector entity table. This table is a connection between the three other tables that were added in ERD 2. It houses not only its own Primary Key, DataConnectorID, to differentiate its own connection from other connections, but as well as Foreign Keys. These Foreign Keys are from the 3 tables it shares a connection with: Start Data, End Data, and Graphs. These foreign keys act as a mediator to show that all three of these tables are connected to one another. This is so that the Graphs table can be used properly.

The main idea behind ERD 2, was not having just a singular Data table like in ERD 1. So instead it was drafted up that ERD 2 will contain instead of the average data over the experiment, it will instead have the starting and ending data from the experiment. All of the data from the start of the experiment will be in the Start Data table, while all of the data from the end of the experiment will be in the End Data table. The data that was stored here is similar to that of Data Table in ERD 1. It will hold data from the three different accelerometers, temperature, pressure, and humidity. The exception is instead of storing the entire time the experiment took place over, the Start Data table will hold the time the experiment stopped, and the End Data table will hold the time it ended. Whereas the Total time the experiment took is stored in our DataConnector table. Of course, this creates a problem, the problem of the user not knowing what the values of the data were any time during the experiment.

To fix this problem, it was thought at the time to also include a graph of the data; this graph was supposed to chart all the data collected from the sensors over a period of time. This Graphs table was in fact supposed to work the same as it would in ERD 1. The only real difference is that the graphs here would be connected to the DataConnector and not the Data itself. As we have two instances of a Data table here I'd need to find a way to connect the graphs with the specific sets of data, to signify that this is where the data for the graphs came from. The DataConnector table was made to try and fulfill this requirement, by having Foreign Keys from all three entity tables.

As mentioned before this is a rough draft and not the final ERD we have decided to model our database off of. The problems ERD 2 faces are the same as to why we didn't use ERD 1. By trying to fix the storing of the data at the start and end of the experiment a new problem arose. The database wouldn't be able to provide the user the necessary data from the experiment anytime during said experiment was happening. Not only that data from the start and end points only may not even be that much of use to the user. The other problem is that the

Graphs table didn't help much either. As mentioned before having the graphs would be nice, but it would complicate matters even more so. The other thing is if you have all the data stored in the database to begin with you wouldn't need the graphs. So, it was decided for the database to function properly, instead of having to store various graphs it would be better to just store all of the data over the course of the experiment. This is what was done in ERD 3.

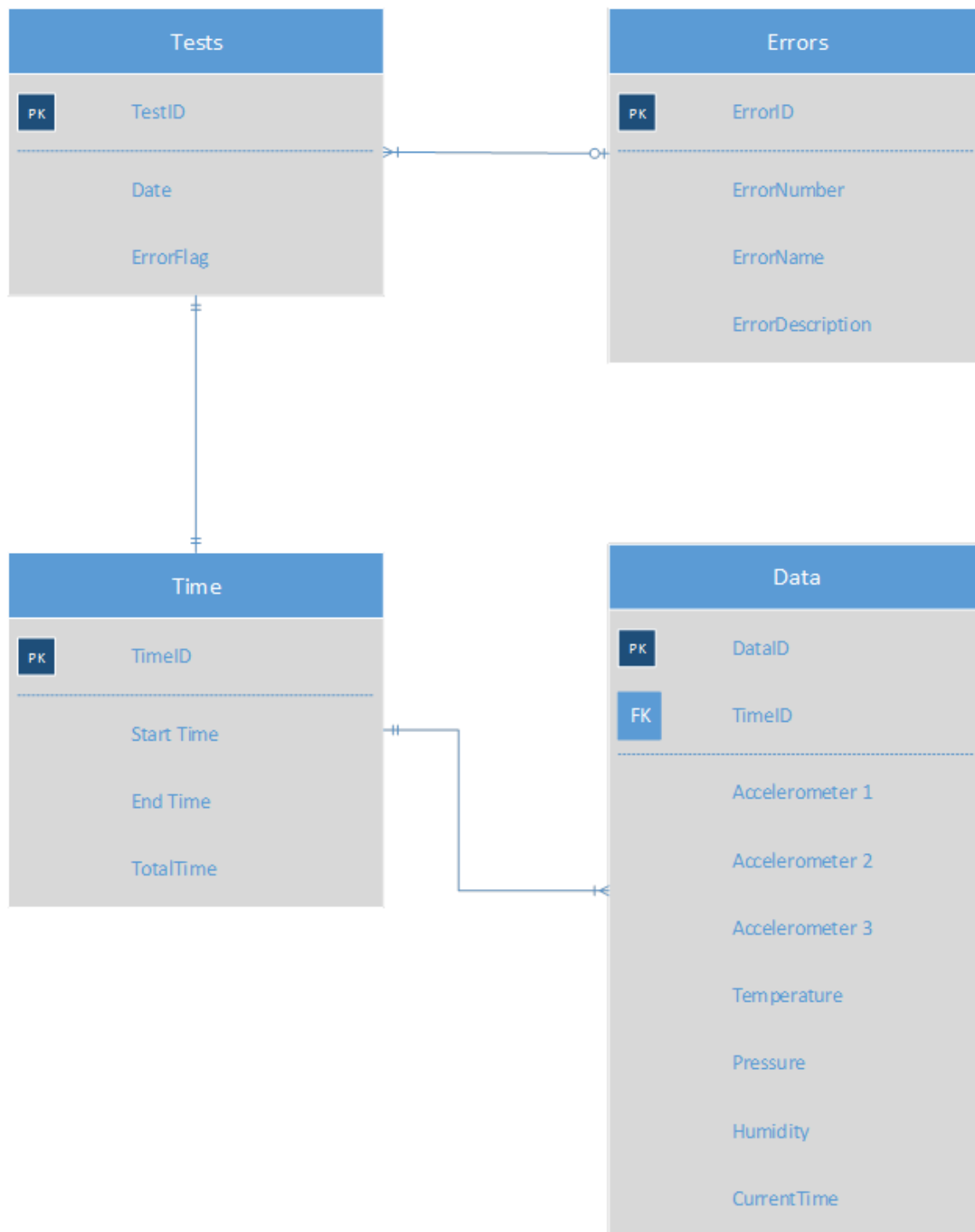


Figure X3 ERD 3

ERD 3 Explanation

The above diagram designated as ERD 3, is the third draft of the database model. ERD 3 has a total of four entity tables: Tests, Errors, Time, and Data. Each of these tables has at least one relationship with each other, and a reason as to why they are made.

To start off the first and primary entity table is referred to as the Tests table. The Tests table shall keep track of very little, but important information. The Tests table will keep track of the Date of the experiment, as well as a variable known as ErrorFlag. The Tests table will also have its own Primary Key (PK in the ERD) known as TestID. The TestID is a unique number given to each experimental test undergone by the reduced gravity experiment balloon. When a new test is ran by the balloon, that test will always have a unique number to differentiate it from other tests. We will also keep track of the Date said experiment was performed on, as it may be important information to retain for each experimental test. Lastly, we will have the ErrorFlag value represent the success of the test. If the test is performed by the balloon without any errors the ErrorFlag will be set to false. But if some sort of error does occur, such as a sensor not work, the ErrorFlag be sent to true. If the ErrorFlag is false there are no errors; while the ErrorFlag is true some error had occurred during the experiment.

The next table is the Errors table. This table similarly to the other tables has its own Primary Key, ErrorID. This is used to identify each error differently from another error that was possible. We will also store the ErrorNumber which represents which error that had occurred, as well the same of that error under ErrorName. Beyond that each error will have its own ErrorDescription, to explain what that error is so the user can try and fix it. The one thing about this table in particular is that it will only be filled out if the ErrorFlag in the Tests table is set to true. Reason being is that if ErrorFlag is false, we don't have an error so there is no reason to establish anything inside this table. So, everything in Error table on an errorless case will be set to 0 and null, to indicate that no errors occurred.

Like previously in ERD 1 and ERD 2, the Tests table and Errors Table perform the very same. The Tests table will have its own unique number, TestsID to differentiate each experimental test from one another. It will continue to store said Date the test had occurred on. The ErrorFlag will still indicate if the Tests had any errors during its experiment. While the Errors table will be filled if and only if an error had occurred during the test. If an error does occur it shall fit out the necessary information, the Error's Number, Name, and the Description of

what that error is. These two tables are to provide the user with some general and necessary information about each experimental test.

Unlike in the first ERD, the Tests table does not directly link to the Data table. Instead it goes to a newly created table called the Time table. This table does not act like an intermediary between the Tests table and Data table. This table holds valuable information about the experimental test itself. Similarly, to other entity tables it has its own unique number, TimeID, to differentiate it from all other experiment tests' times. It holds the Start Time of the experiment test, the End Time of the experiment test, as well as the Total Time the experiment underwent. This information is necessary as will inform the user when the experiment took place. But it will also help illustrate when the data is collected and how it is displayed in the Data table.

The Data table in ERD 3 is similar to the other ERDs, as it also stores the same type of data. It will hold data from the three different accelerometers, temperature, pressure, and humidity. It also holds the data of the specific time that this data was collected (called CurrentTime), which will always be between the Start Time and End Time. Unlike before though, this is not merely a singular entity table. The Data table and the Time table share a many to one relationship. It is required that the Time table has to have at least one or more Data table connected to it. While the Data table has to belong to specifically one Time table. This allows for us to store multiple Data tables by having them all linked to the same Time table if they belong to the same experimental test. This relationship between these two tables allows us to store all our data of the experiment over the time it took. If we were to say the experiment ran for 30 seconds, and we collected the data from the sensors every second, we would have a total of 30 Data tables that all link back to our Time table. This fixes our biggest problem of not storing the data from the experiment over the course of time. The last thing to mention about the Data table is why it has the Foreign Key of the Time table it belongs to (called TimeID in the ERD). Since we will be storing data from the sensors constantly for each test, it will result in a large number of Data tables over the course of multiple experiment tests being run. So, to help organize it, the Data tables will have this Foreign Key to help identify where it belongs in the grand scheme of things. Its other purpose is to make sure that the data the was collected is within the time period of said experiment. The CurrentTime is stored in the Data table for when that data is collected, to make sure that this Data table did happen during the experiment we will need to keep hold of the times of the experiment. This will help in identifying if the data collected is within the time period of the experiment being ran.

As mentioned before this is a rough draft and not the final ERD we have decided to model our database off of. Originally before beginning the project this was the final draft of the ERD of how database system was to be made. Since the start of work on the project, we have hence decided to use authentication in our project. We also wanted to store more information on the sensors as well, such as the different types that used. We also needed the ability to have a way of detecting errors based on the sensors that are used besides just being used as a link to the Tests table. Thus, when designing the final form of the ERD for the database we decided to include these things and change how the relationships for the tables already existed.

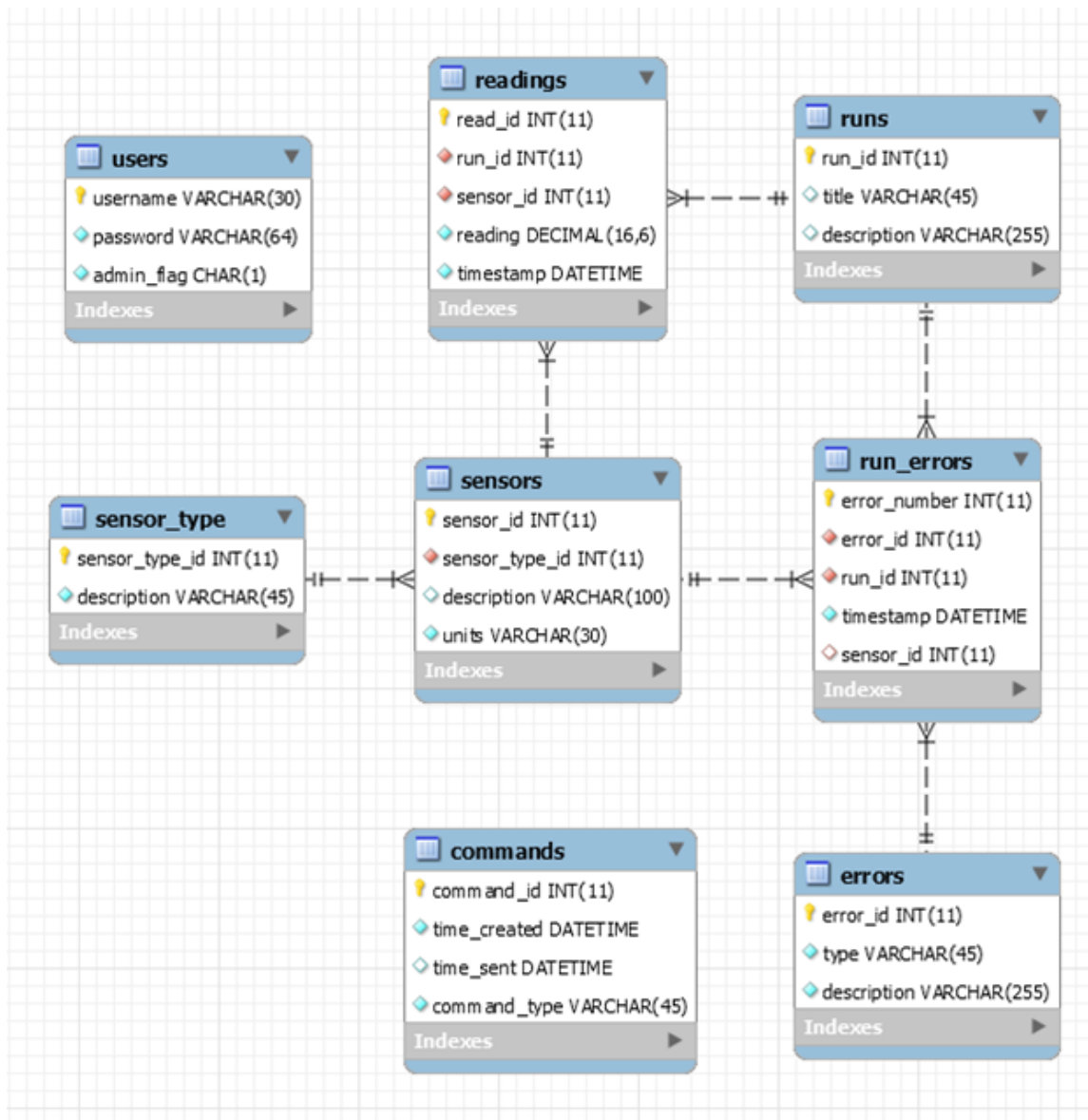


Figure X4 ERD 4

ERD 4 Explanation:

The above diagram designated as ERD 4, is the final version of the database model. This is the model of the database that is used for the entirety of the project. ERD 4 has a total of eight entity tables: runs, run_errors, errors, readings, sensors, sensor_type, users and commands. Also, this ERD is specifically made in the MySQL workbench which is what used for our designing and creation of the database.

Unlike the previous ERDs, the tables in the final version have completely different table names, but many act and behave in the same way. Such as runs table being very similar to the Tests table, as well as the errors table beings similar to the obvious Errors table. The reading table acts like the data table in the previous ERDs.

To start off the first and primary entity table is referred to as the runs table. The runs table shall keep track of very little, but important information. The runs table keeps track of the title and description of the run. It also has its own Primary Key (The yellow key in the ERD) known as run_id. The runs will keep track of the readings (readings table) that are associated with that specific run, as well as any errors that are associated with that specific run.

The next table is the run_errors table. This table acts as an intermediary between three tables, the runs, sensors and errors tables. It has its own Primary Key, error_number as well as three foreign keys (the filled and unfilled red diamond). These foreign keys are the error_id, the run_id and the sensor_id, which are the primary keys from the errors, runs, and sensors table. It has a timestamp value which keeps track of when the run_error occurred during each run of the tests. As when an error occurs we would want to know in which run it happened in, as well as (when applicable) which sensor caused it. We would then want to know the specific error that occurred. So, the errors table is the table that lists the errors that can occur. Having it be related to the run_erros is needed as the run_errors will tell us which error occurred during which test.

The next table is the readings table. This is similar to the Data tables in the previous ERD as it holds the data from our various sensors. It, like the other tables has its own Primary Key as read_id. It also has the foreign keys from runs and sensors tables as run_id and sensor_id respectively. It needs this because each set of reading has to correspond to a specific run it belongs to, as well as to the sensor the data reading came from. Inside the table it also holds the reading

data value as reading as well as the timestamp the data was collected on the balloon.

The sensors and sensor_type table go hand in hand together. The sensor_type table is to store the data on the various different type of sensors we have. As it has its own Primary Key as sensor_type_id as well as a description of what the sensor type collects. The sensor table though has the foreign key from the sensor_type table. This is because we could (and we do) have multiple instances of the same type of sensor. In our project we have multiple accelerometers. So, the sensors table has its own unique Primary Key as sensor_id to differentiate each specific sensor we have. It also has a description of it, as well as the units the sensor is measuring. Because each sensor could possibly measure something in different units.

The last two tables are the users and the commands table. Both are used for specific situations in the database. The commands table is used from the frontend of the website to send specific commands to the balloon. Such as starting and stopping a test for example. This has its own unique primary key, as command_id. It also has the time the command was created and the time it was sent to the balloon. As well as what specific command type that was issued. The users table is built in for functionality of authentication on the frontend. This is where we will store all the users who are a part of the system. Only users who are a part of the system can see and view data. Each username is the unique Primary Key of that said user. The password is also stored here except it is the hashed password stored from the frontend application. There is also an admin flag that determines whether or not you are an admin. As an admin determines whether or not you can perform certain commands on the front end.

This is the database diagram that was used for our database in this project. It includes the storage of the data from the sensors. It also stores the errors that could potentially occur from the balloon. It even has the commands that can be sent to and from the front-end web to be sent to the balloon. And lastly it has the necessary information of storage for the use of authentication. This is what we have decided that the database model is in MySQL.

Front End Display

In an overview, the front end will be an AngularJS app running on a Node web server using Express. As a result of using these technologies, the server is easy to get setup and very versatile with the ability to adapt to changes as needed. Node and Express make it possible to establish an API with which to store and retrieve data from that database with ease. It was our goal to take this project in a direction that could easily be extended to fit the needs of anyone using this technology. In the following section, we will cover the setup and structure of the application and its interfaces with the database.

Design Details

Node.js + Express

Especially when used with the rest of the MEAN stack, Node.js + Express is ideal for the creation of a single highly-interactive web page. However, it is equally useful for a multi-page web application like ours.

This web application must rely on dynamic web pages that change instantaneously when a specific change is made. Because Django has a synchronous protocol and would thus be ill-suited for a dynamic web application like this one, Node.js + Express is the obvious choice with its asynchronous protocol.

AngularJS + Chart.js

When combined with the rest of the MEAN stack, AngularJS is ideal for creating a dynamic web page. The two-way data binding in particular helps greatly in this regard. Because the web application for this project must rely on a dynamic web page for displaying rapidly changing data, AngularJS is used for its creation, as the usefulness of AngularJS greatly outweighs the high learning curve.

In addition to a standalone web page that is dynamic by itself, a dynamic data tool is also needed to display the data in a chart, and Chart.js serves this purpose. Although it does not offer many features, the features it does have suit the purposes of this project. The simple charts provide visually appealing charts that update quickly, which is the most important quality needed for the web page. Google Charts would not have been fitting to use, simply because a constant, reliable internet connection could not be guaranteed during any runs for experiments.

In order for dynamic charts of data to be implemented in the web page, Chart.js and AngularJS must work together properly. While the external package angular-chart.js would have been useful in accomplishing this, it has not been documented and updated enough to be particularly useful. Therefore, it was decided that we would make do without it.

Web Application

For the purposes of this project, the web application will be able to manage and display large amounts of data on the page dynamically. Information from each of the sensors on board the capsule can be displayed at the same time, and settings for changing different aspects of the capsule or application can also be managed.

When a user first navigates to the website, they are redirected to the login page in Figure 8. Until the user logs in, they will be unable to see any other part of the application.

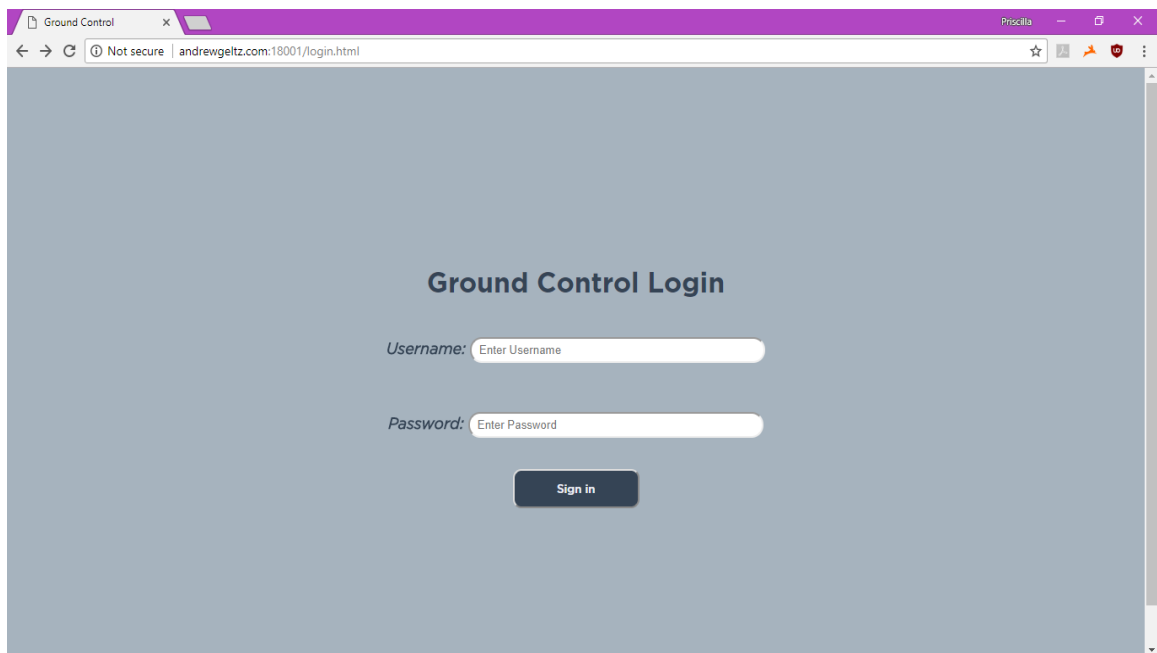


Figure 8 Login Page

Once logged in, the user is redirected to the index page in Figure 9, where they will have the option of starting the test and having the live run display on the page, stopping the currently running test, or simply displaying the results of the most recent run. Alternatively, the user may also click on the hamburger menu in the upper left corner of the page to display different settings pages, along with a link to a history of previous runs.

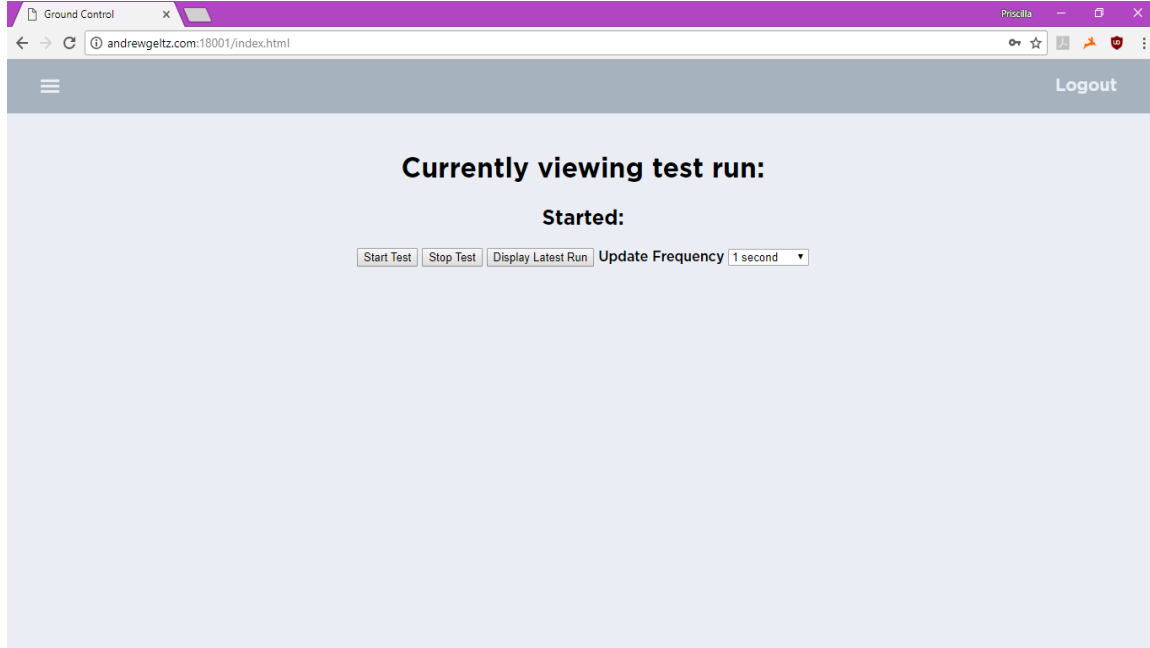


Figure 9 Index Page

Individual charts in the web application will contain time in seconds on the x-axis, while the y-axis will contain whichever variable is suitable to the type of measurement in that graph. Every sensor that is set to measure data on the capsule will display data as a separate chart on this index page, with each one being labelled with its identifying name. An example chart from a live data run can be seen in Figure 10.

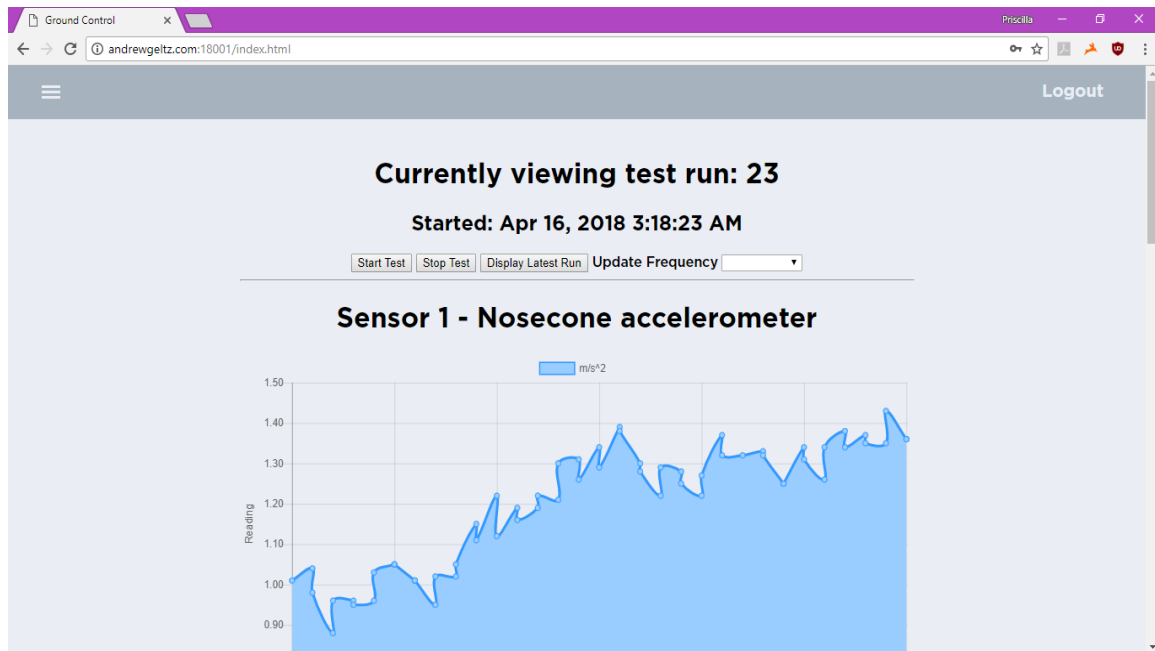


Figure 10 Index Page with Chart

For settings, a user can click on the hamburger menu to bring up a settings menu, visible in Figure 11. The user can view historical runs and also manage errors received from the capsule, sensors being used during runs, and users registered in the database. Alternatively, the user may also log out of the web application at any time by clicking the “Logout” button in the upper right corner of the web page.

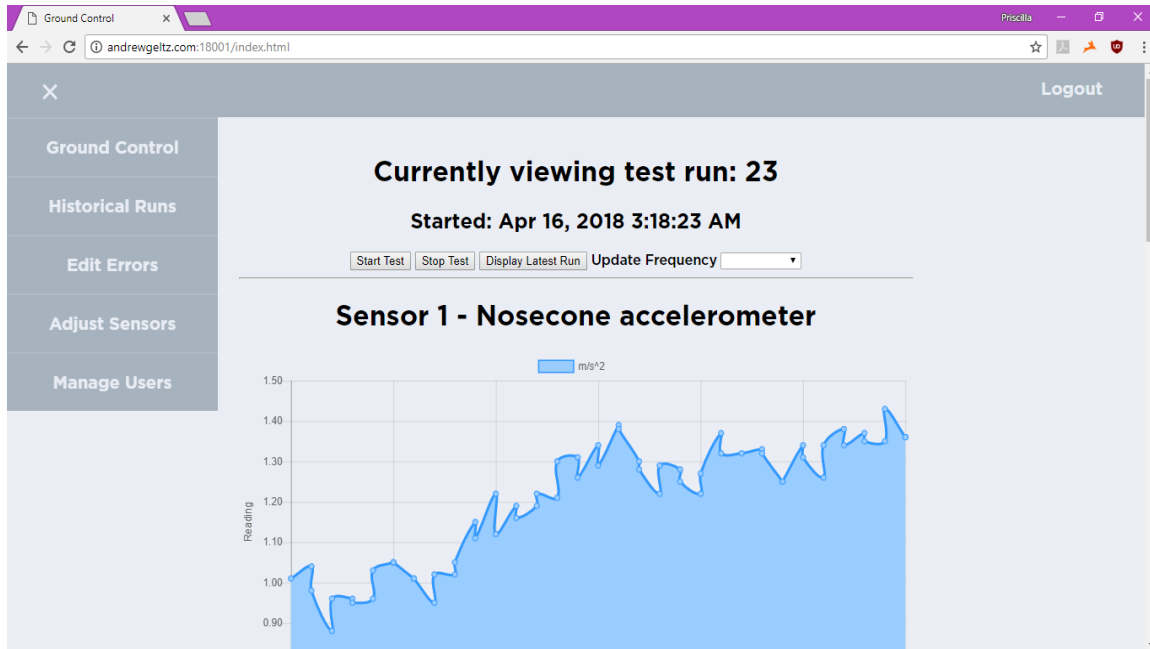


Figure 11 Index Page with Hamburger Menu

Should a user click on “Historical Runs,” they will be brought to the page of the same name in Figure 12. Clicking on “Edit Errors,” “Adjust Sensors,” and “Manage Users” will lead to the pages in Figures 13, 14, and 15, respectively.

The Historical Runs page displays a simple table filled in with data from previous runs. Each run consists of a Run ID, Start Time, Run Title, and Run Description.

Select Runs to Display

Run Id	Start Time	Run Title	Run Description
3	Apr 15, 2018 7:29:44 PM	Demo run	Demo run description
5	Apr 15, 2018 7:41:50 PM	Demo run	Demo run description
6	Apr 15, 2018 8:00:53 PM	Demo run	Demo run description
7	Apr 15, 2018 8:08:43 PM	Demo run	Demo run description
8	Apr 15, 2018 8:21:32 PM	Demo run	Demo run description
9	Apr 16, 2018 1:31:59 AM	Demo run	Demo run description
10	Apr 16, 2018 1:44:32 AM	Demo run	Demo run description
11	Apr 16, 2018 1:46:20 AM	Demo run	Demo run description
12	Apr 16, 2018 2:06:31 AM	Demo run	Demo run description
13	Apr 16, 2018 2:08:14 AM	Demo run	Demo run description
14	Apr 16, 2018 2:14:03 AM	Demo run	Demo run description
15	Apr 16, 2018 2:15:45 AM	Demo run	Demo run description

Figure 12 Historical Runs Page

Errors

Current Errors

Error Id	Error Type	Error Description	Delete?
1	Non-critical sensor	Unable to read barometer	Delete
2	Fatal systems problem	Power supply insufficient!	Delete

New Error

Error Type: Error Description:

New Error

Figure 13 Errors Page

The Errors page displays a table consisting of all current errors output by all currently used sensors. Each error consists of an Error ID, Error Type, and Error Description, and the user may delete a given error at any time by clicking the “Delete” button next to that error. Additionally, the user may add a new error into the database utilizing the form beneath the table; filling in the required information and then clicking “New Error” will submit to the database and display back to the page.

The Sensors page displays two tables for the user: one for all current sensors, and one for all current sensor types. Each sensor consists of a Sensor ID, Sensor Type, Sensor Description, and Units recorded in, and the user may delete a given sensor at any time by clicking the “Delete” button next to that sensor. Additionally, the user may add a new sensor by utilizing the form above the table; filling in the appropriate values and then clicking “New Sensor” will submit to the database and display back to the page.

The Sensor Types table is structured similarly. Each type consists of a Sensor ID and Sensor Type Description and may be deleted at any time with the “Delete” button. A new one may be added with the form above the table as before.

Sensors

Sensor Type: Sensor Description: Units:

Sensor Id	Sensor Type	Sensor Description	Units	Delete?
1	1 : accelerometer	Nosecone accelerometer	m/s ²	<input type="button" value="Delete"/>
2	1 : accelerometer	Tail accelerometer	m/s ²	<input type="button" value="Delete"/>
3	2 : thermometer	Experiment compartment thermometer	c	<input type="button" value="Delete"/>
4	3 : barometer	Experiment compartment barometer	atm	<input type="button" value="Delete"/>

Sensor Types (Groups)

Sensor Type Description:

Sensor Id	Sensor Type Description	Delete?
1	accelerometer	<input type="button" value="Delete"/>

Figure 14 Sensors Page

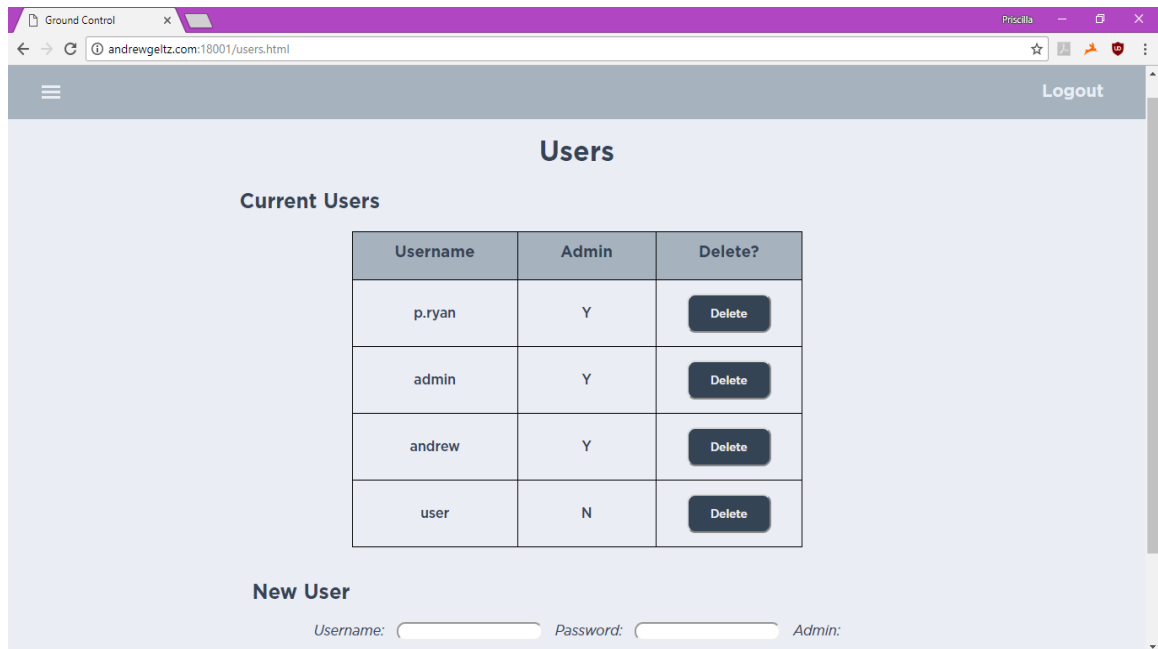


Figure 15 Users Page

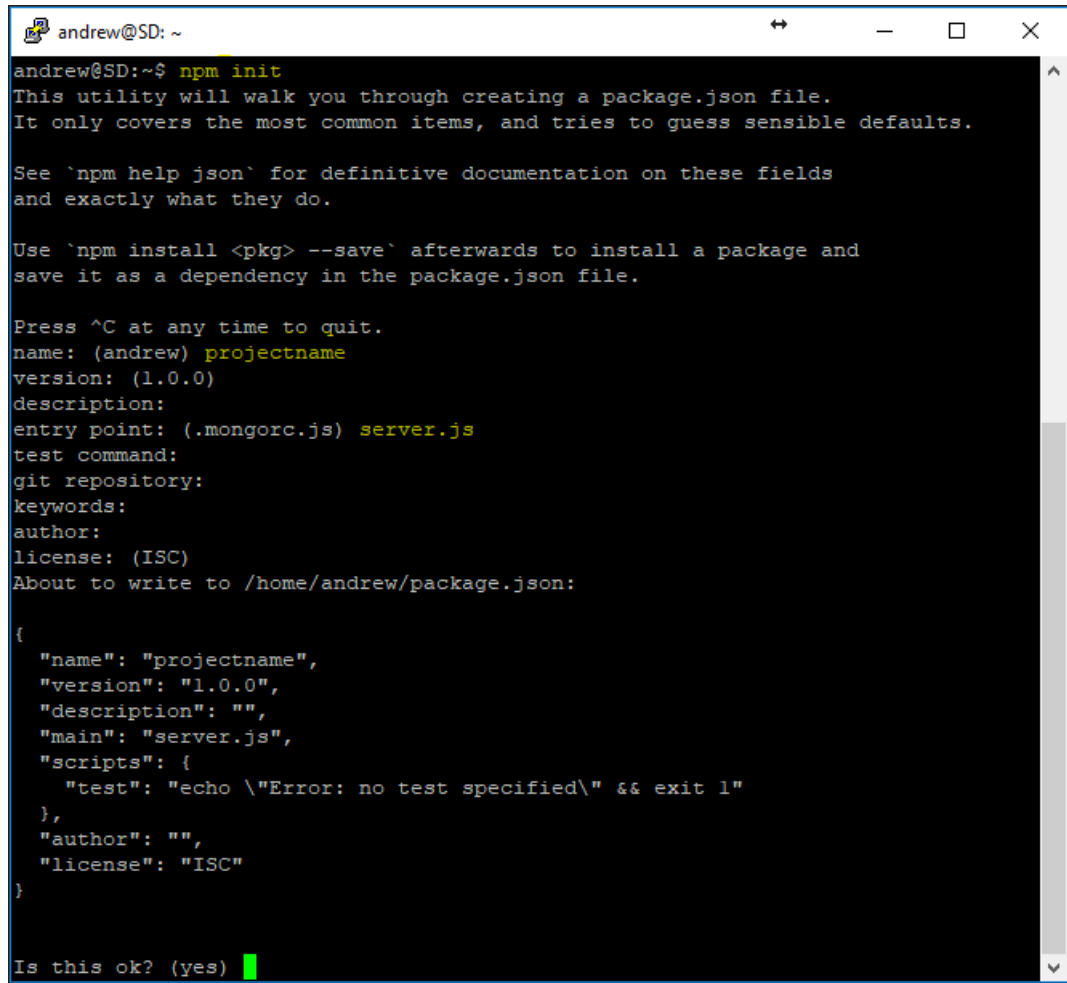
The Users page, like the previous two mentioned, displays a table of all registered users, along with a form at the bottom for adding a new one. Each user entry consists of a Username, Password, and Admin flag, although the password is not displayed in the table for security and privacy reasons. An admin may delete any other user using the “Delete” button next to that user’s entry.

Initial Server Setup

The following section details the initial setup and configuration of the web server and its component parts. A simpler version that makes use of the Docker container and setup scripts will be in the following section. This section will detail the steps that were taken to initially setup the application. It is assumed that the server has been installed and configured with Ubuntu Server 16.04. A valid internet connection is required to install many of the modules. Many steps in the following do require sudo access which comes with its own dangers if commands are mistyped, so a certain level of vigilance is required. All commands to be entered are preceded with a \$ and in red text.

1. Make sure that the system is up to date by running the following commands. The update command pulls the latest package information from the Ubuntu servers (or other servers if the server is configured as such). After the package information is updated, the upgrade command uses the new information and fetches any required updates for all programs that are installed on the machine. The upgrade command will list the packages to be updated and the required disk space to install the updates. When prompted, type y and press enter to acknowledge the updates.
 - a. `$ sudo apt-get update`
 - b. `$ sudo apt-get upgrade`
2. Install Node and Mongo onto the server. After each command, the system will tell you what dependencies are required and how much space the installed program will take up. Acknowledge this when required by typing y and pressing enter. This may take a few minutes depending on the internet connection and the configuration of the server.
 - a. `$ sudo apt-get install nodejs npm`
 - b. `$ sudo apt-get install mysql`
3. Now we will be creating the directory where the Node application will reside.

- a. Make the directory. We chose to make this at the root of the disk to make paths shorter when using fully qualified paths. This also allows all users to access it which is good because the only users on this machine are going to be the project team.
`$ sudo mkdir /project`
 - b. The directory had to be created as root (using sudo) because it is a top-level folder. This created the directory with very strict permissions that would interfere with the operation of Node as it is intended to only be ran with user level commands. There are ways to run it as root but it is simpler to change the permissions on the folder. The chmod command modifies the permissions of folders and files. The a+rw option means that we want to modify permissions for all users (a) by adding (+) read and write (rw) permissions.
`$ sudo chmod a+rw /project`
4. The next step is to configure the basis for the Node application with a package file. Node can generate one for us and that is what we will do but it should be known that the package.json is a simple json file that can be edited without any special tools if changes need to be made after the initial setup.
- a. Navigate to the project directory.
`$ cd /project`
 - b. Start the project generator.
`$ npm init`
The following screenshot shows the prompts that the generator will



```
andrew@SD: ~  
andrew@SD:~$ npm init  
This utility will walk you through creating a package.json file.  
It only covers the most common items, and tries to guess sensible defaults.  
  
See `npm help json` for definitive documentation on these fields  
and exactly what they do.  
  
Use `npm install <pkg> --save` afterwards to install a package and  
save it as a dependency in the package.json file.  
  
Press ^C at any time to quit.  
name: (andrew) projectname  
version: (1.0.0)  
description:  
entry point: (.mongorc.js) server.js  
test command:  
git repository:  
keywords:  
author:  
license: (ISC)  
About to write to /home/andrew/package.json:  
  
{  
  "name": "projectname",  
  "version": "1.0.0",  
  "description": "",  
  "main": "server.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC"  
}  
  
Is this ok? (yes)
```

Figure 14 Npm init generator

ask and I have highlighted the things that I typed.

The project name does not really matter but it is a good choice to pick something meaningful. The main important section of this setup is the entry point. This is the file that Node looks at to start the project. A common name for this file is index.js but we have opted for server.js as it makes the purpose of that file clearer. The other options are entirely optional. Once the options are entered, the generator will ask if that configuration file is okay. Press enter to accept the parameters or Ctrl + C to exit and try again.

5. Now that the project is established, we will install several Node modules to give us functionality and aide in the development process. For installing these packages, we will be using the Node Package Manager, or npm.

The extra option that we are including on the end of the install command, the `--save`, tells npm to update the package.json and mark that module as a required dependency for our project. By default, it allows the project for use with any versions of the module at the version we are installing or newer which works just fine. All of these packages will be installed in a node-modules folder from the directory you run the command in. In our case, the modules will be installed in /project/node-modules

- a. Express (<https://expressjs.com/>) is a very popular framework for Node to handle many of the complicated aspects of the server side of the application. In our case, it is what runs the actual web server on top of the main Node framework.
`$ npm install express --save`
- b. MySQL driver (<https://www.npmjs.com/package/mysql>) is a tool that allows the application to run queries against the MySQL database.
`$ npm install mysql --save`
- c. Body-parser (<https://github.com/expressjs/body-parser>) is a Node module that allows Node to be able to parse json into javascript data objects. This module makes sending data to and receiving data from the database much easier.
`$ npm install body-parser --save`
- d. Morgan (<https://github.com/expressjs/morgan>) is an extension module for Express that displays http requests in a human readable format as the server is running. This module makes it easier to see the requests as they come in and troubleshoot any issues. Note that we are not using the `--save` option because we do not want to add Morgan as a dependency for the project as it is more of a development tool.
`$ npm install morgan`
- e. Nodemon (<https://github.com/remy/nodemon>) is a tool that allows the Node application to detect any changes in the files and restart the application. This allows for rapid development and any changes only require refreshing the browser page. We are going to install this as sudo and as a global package with `-g` so that we can call it

from a bash script to autostart the server.
`$ sudo npm install nodemon -g`

- f. Bcrypt (<https://www.npmjs.com/package/bcrypt-nodejs>) is an encryption library that will handle hashing of passwords for the authentication done by passport.js. One of the main draws of bcrypt is its ease of use in implementation.
`$ sudo npm install bcrypt-nodejs --save`

- g. Passport.js (<https://www.npmjs.com/package/passport>) is a framework that abstracts authentication into an easy to use package. It supports many type of authentication such as local (stored in your database), facebook, google, and Oauth. It also provides support for securing APIs. Since we are only using the local authentication, we only need to install the core package and the local module.
`$ sudo npm install passport --save`
`$ sudo npm install passport-local --save`

- 6. Now that we have all the necessary modules installed, we need to create our main Node file, server.js. The following sections break down the lines and what they do.

- a. Imports

```
// Imports
var express = require('express');
var app = express();
var bodyParser = require('body-parser');// Used for parsing json
var mysql = require('mysql');
var router = express.Router();
var path = require('path');
var passport = require('passport');
```

Figure 15 Node application imports

The imports at the top of the file allow us to access code from other files and from the modules we installed. Unlike some other languages, Node stores the imported code into variables with the require function. This function does the double duty of ensuring dependencies are met, as well as, making the code available to the

program. The app variable is main controller for the whole application. The router and appRoutes variables are the imports for the database api setup later. The path variable is used to get the current path which is used to reference the project path later in the file.

b. Variables

```
// Variables
var port = process.env.PORT || 8080;
```

Figure 16 Node application variables

Currently there is only one configuration variable and that is the port the server will listen on. The port variable will be set to either the environment variable for PORT or 8080 if one is not defined.

c. Middleware

```
// Middleware - Order matters!
app.use(require('morgan')('dev')); // Do verbose console logging
app.use(require('cookie-parser')());
app.use(require('express-session')({ secret: 'super secret session', resave: false, saveUninitialized: false }));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Figure 17 Node application middleware

The middleware is the software calls before the main app. In these cases, we tell the Express app that we will be using Morgan, body-parser's json module as well as the api calls we imported into appRoutes. We also tell Express that we want the api to be available on the server at /api (ie. serveraddress/api). This allows us to separate the server-side and client-side aspects of the application.

d. Database

Connection

```
// Database Connection
var databaseConfig = require('./config/database'); // Pull connection string from config
dbConnection = mysql.createConnection(databaseConfig.connectionString); // Create connection from string
dbConnection.connect(); // Initiate connection
```

Figure 18 Node application database connection

Using the MySQL module that we imported, we attempt to connect to the database `ground_control` on the local server using a connection string that we are loading from another file (`config/database.js`).

```
// Initialize Passport and restore authentication state, if any, from the session
app.use(passport.initialize());
app.use(passport.session());
require('./config/passport')(passport);
```

e. Passport initialization

We tell the application (Express) that we want to start up the passport handlers. We also load in our passport config which outlines how to handle authenticating users.

f. Web Routes

```
// Api setup
app.use('/api', router); // Serve the api off of ip:port/api/...
require('./app/controllers/api')(router, passport); // Import our API

// Setup routes
require('./app/routes')(app, passport, __dirname + '/public/');
```

Figure 19 Node application web routes

We tell the Express app to load all of our api routes that define our REST api by passing the express router to our main api handler in the controllers folder. The next step is to handle the non-api routes which is done in the routes file in the app folder. We also pass the passport module so that we can authenticate requests.

g. Starting the Server

```
// Start server
app.listen(port, function() {
  console.log('[SERVER] Ground Control waiting for Major Tom on port: ' + port + '!');
});
```

Figure 20 Node application server start listening

We now tell the express app to listen on the port specified and as soon as this call returns (ie the server starts listening), we display a message to the console so that we know the server is up and running.

```
// Imports
var express = require('express');
var app = express();
var bodyParser = require('body-parser');// Used for parsing json
var mysql = require('mysql');
var router = express.Router();
var path = require('path');
var passport = require('passport');

// Variables
var port = process.env.PORT || 8080;

// Middleware - Order matters!
app.use(require('morgan')('dev'));// Do verbose console logging
app.use(require('cookie-parser')());
app.use(require('express-session')({ secret: 'super secret session', resave: false, saveUninitialized: false}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Database Connection
var databaseConfig = require('./config/database');// Pull connection string from config
dbConnection = mysql.createConnection(databaseConfig.connectionString);// Create connection from string
dbConnection.connect();// Initiate connection

// Initialize Passport and restore authentication state, if any, from the session
app.use(passport.initialize());
app.use(passport.session());
require('./config/passport')(passport);

// Api setup
app.use('/api', router);// Serve the api off of ip:port/api/...
require('./app/controllers/api')(router, passport);// Import our API

// Setup routes
require('./app/routes')(app, passport, __dirname + '/public/');

// Start server
app.listen(port, function() {
  console.log('[SERVER] Ground Control waiting for Major Tom on port: ' + port + '!');
});
```

h. Entire server.js

Figure 21 Node application entire server.js

7. Now that the server.js application is setup, we will now work on making it start with the server as a nodemon process. Nodemon will watch for any file changes and restart the server process as required so we don't have to stop and restart the application every time we make a change. Normally

we can start the server with nodemon by simply executing nodemon server.js. However, we want this to happen automatically when the server starts and without blocking up our terminal. We will be using the screen application which is installed with Ubuntu Server by default to start the nodemon process in its own terminal. This has the benefit of allowing us access to the command line while it is running but also lets us check in on the command line output of the server by opening the process's screen whenever we would like.

- a. We will start by creating a bash script that will start the server application with nodemon.

```
$ nano autostart.sh
```

- b. When the file is open, type the following lines. The first line tells the operating system that this file should be executed as a bash script and the second line is the command to be executed.

```
#!/bin/bash
```

```
nodemon /project/server.js
```

- c. Once the two lines have been added to the file, close nano with

```
Ctrl + X
```

```
Y
```

```
Enter
```

```
Enter
```

- d. Now we need to mark this file as an executable.

```
$ chmod u+x autostart.sh
```

- e. To make the file execute when the server starts, we will add a command to the rc.local file.

```
$ sudo nano /etc/rc.local
```

- f. Anywhere before the line that says exit 0, add the following command. This command starts a new (-m) detached (-d) screen session called nodeapp with the shell script (-s) located at /project/autostart.sh

```
screen -d -m -S nodeapp /project/autostart.sh
```

- g. Close the rc.local file.

```
Ctrl + X
```

Y
Enter
Enter

- h. Restart the server to make sure that the script runs successfully. To access the screen session that is created we used the resume (-r) option and the session name.

```
$ screen -r nodeapp
```

If for some reason, there is more than one session called nodeapp, the sessions will be listed and prepend the session id to the name.

```
$ screen -r pid.nodeapp
```

Once in the screen session, it can be killed with

Ctrl + A, K (Control and A at the same time and then press K)

The screen session can be detached, exited without stopping the session with

Ctrl + A, D

Additional information about the screen application can be found in the linux man pages.

```
$ man screen
```

Quick Start Guide

Being portable and easy to use was one of the main tenets of the design plan for this project. To help accomplish this, we built a docker container to run the application and created a few scripts to help get the application up and running as easily as possible. The following section describes how to utilize these tools. Lines starting with \$ and in **red font** denote commands to be entered in the command prompt

1. Clone the GitHub repository
(<https://github.com/ucfmae/MicrogravityCSSp18>)
\$ git clone https://github.com/ucfmae/MicrogravityCSSp18.git
2. Install Node, Docker and MySQL
\$ sudo apt-get install nodejs
\$ sudo curl -fsSL https://get.docker.com | sh
\$ sudo apt-get install mysql
3. Create a new user in the mysql application and make sure to allow them to connect from remote hosts as well because the docker container has its own ip (<https://www.digitalocean.com/community/tutorials/how-to-create-a-new-user-and-grant-permissions-in-mysql>)
4. Run the schema.sql script to setup the database and all the stored procedures (where [username] is the user account you set up or the root account). You must be in the root folder where the schema.sql file is.
\$ mysql -u [username] -p < schema.sql
5. Build the docker container and call the built container ground-control. You must be in the directory with the Dockerfile
\$ docker build -t ground_control .
6. Start the docker container, run a docker container detached (-d) and expose port 8080 (-p {host port}:{container port}) and name it ground_control using the ground_control image we created earlier.
\$ docker run -d -p 8080:8080 --name ground_control ground_control
7. Verify the container is up and running (the -a shows all containers including stopped ones)
\$ docker ps -a

8. To stop the application (since we named it ground_control, we can stop it by name)
`$ docker stop ground_control`
9. To remove the stopped container
`$ docker rm ground_control`

We have also included a demo interface application for testing purposes. There are two python programs in the interface directory. The first, setupDemoDatabase.py, removes all data from the tables and inserts some generic sensors. The second, demo.py, waits for a start command from the interface and then dumps 30 seconds worth of fake sensor data into the database. Also, after 100 readings, the demo script generates an error.

The general process for the demo is:

1. Start the application via node or the docker container
2. Run
`$ python interface/setupDemoDatabase.py`
3. Run
`$ python interface/demo.py`
4. Click on the 'Start Test' button on the home page after logging in.

Project File Structure

In the following list, folders are **RED**, javascript files are **BLUE** and other files are **GREEN**.

1. **app** - Stores entire server-side application, files are imported into the server.js
 - 1.1. **routes** - Stores the Express routes that establish the API
 - 1.1.1. **api.js** - The main API file for Express, includes all the files in app/models to generate the data structures to send to the database
2. **node-modules** - Stores all Node modules installed with npm
3. **public** - Stores all client-side files
 - 3.1. **controllers** - Stores all Angularjs controllers to be imported into public/app/app.js
 - 3.1.1.
4. **autostart.sh** - The bash script that is called from /etc/rc.local to run our Node application at startup
5. **package.json** - The Node application package variable, this includes the dependencies and project name
6. **server.js** - The entry point for the server and the main Node files

Project Milestones

- Project Requirements Document - October 9th 2017
- Project Status Update With TA - Week of October 30th 2017
- Project Status Update with Professor - Week of November 11th 2017
- Semi-Final Design Choices - November 15th 2017
- Turn in pages completed - November 17th 2017
- Goal - Get Required Services in Place - November 22nd
- Goal - Initial Handshake Between Services - November 29th
- Final Design Document - December 4th 2017
- Proof of concept for Mechanical Engineering teams - End of Fall 2017
- Critical Design Review - February 2018
- Heinrich Demo - March 2018
- Final Presentation - April 2018

Build, Prototype, Test, and Evaluation Plan

Build Plan

As was discussed in the Implementation Details section, most of this project will be built primarily using the MEAN stack. MySQL will be the database storing all of the data from the sensors on board the capsule, while Node.js + Express handles the exchange of information between devices. AngularJS, along with Chart.js and other APIs, will help format the data in such a way that is useful to users.

Following along with all of the aforementioned plans and steps, we will attempt to build a prototype that meets all of the requirements laid out by the original problem. We plan to follow the information in this document as closely as possible, but should difficulties or better alternatives arise during development, we will carefully evaluate them and decide if certain design aspects should be changed or not. If they should, then the design will be modified to fit the new addition to the plan.

Prototype

The planned prototype for this project will consist of a mock capsule transmitting random or predetermined data to the Raspberry Pi, which will then forward the data to a second device that will be running the core software for the program. The device will communicate information to the online MongoDB database through Node.js, which will also in turn be communicated to the main web application as well. The web page for the application will receive the data in real time and also display it to the user in real time, dynamically updating the interface as the prototype trial progresses.

Test Plan

The test plan for the prototype will consist primarily of individual testing of different components of the entire program as development progresses; individual components must work by themselves before they can be expected to work with others. Communication between the different components will also be tested early on to ensure that test results down the road are not influenced by faulty communication design.

Once core development has finished, all aspects of the prototype will be tested simultaneously in order to find any remaining bugs in the programs. If bugs are found at this stage, testing will return to the individual component stage until they are believed to have been removed.

Evaluation Plan

Once all development has been completed, the final product will be evaluated for its adherence to the original requirements laid out in the Project Requirements section, and for its adherence to the basic Human-Technology Interaction principles. While accounting for the major requirements of this project is the main priority, traits like learnability and usability are still very important to the future of this project.

In particular, speed of the web application will be a very important factor in determining the overall success of the project. The core of the project relies on quick data transmission for near-instantaneous results on the web page that users can look at, so that they may quickly discern important data as it is being gathered. Thus, speed of the application will be a critical evaluation factor.

Budget and Funding

Quantity	Price per unit	Description	Subtotal
2	\$70	Raspberry Pi 3 kit (includes power supply and SD card)	\$140
2	\$5	NRF24L01+ RF Transceiver	\$10
1	N/A	Dell R610 Server	N/A
1	\$10	Cat 6 ethernet cable 5 pack	\$10
		Total	\$160

As the costs were minor and we utilized equipment that was available to us, Andrew offered to cover the expenses out of pocket.

References

Figures

- Figure 1 - Proposed Solution
- Figure 2 - NRF24L01+ - (https://hackspark.fr/pub/media/catalog/product/cache/image/700x700/9b53d7ee6c576e27421bdbaefdf2e7a2/i/m/im120606003_10.jpg)
- Figure 3 - Communications Considerations
- Figure 4 - Raspberry Pi 3 Model B - (<https://www.raspberrypi.org/app/uploads/2017/05/Raspberry-Pi-3-hero-1-1571x1080.jpg>)
- Figure 5 - Dell PowerEdge R610 - (http://i.dell.com/das/xa.ashx/global-site-design%20WEB/0ac50c57-a621-67ee-6313-347bb05abaaa/1/OriginalPng?id=Dell/Product_Images/Dell_Enterprise_Products/Enterprise_Systems/PowerEdge/PowerEdge_R610/hero/server-poweredge-r610-left-hero-504x350.png)
- Figure 6 - Raid 1 - (https://www.seagate.com/files/www-content/manuals/business-storage-nas-os-manual/shared/images/117_ill_raid_1.png)
- Figure 7 - DrivePool logo - (<https://stablebit.com/Content/Images/hbar/large/drivepool/logo.png?2>)
- Figure 8 - Rough layout of web page
- Figure 9 - Example graph using chart.js
- Figure 10 - Sensor checkboxes
- Figure 11 - Run checkboxes
- Figure 12 - Web page after sensor checkbox update
- Figure 13 - Web page after run checkbox update
- Figure 14 - Npm init generator
- Figure 15 - Node application imports
- Figure 16 - Node application variables
- Figure 17 - Node application middleware
- Figure 18 - Node application database connection
- Figure 19 - Node application web routes
- Figure 20 - Node application server start listening
- Figure 21 - Node application entire server.js
- Figure 22 - NRF24L01+ to Raspberry Pi pin connections

Software

- MySQL - (<https://www.mysql.com/>)
- Express.js - (<https://expressjs.com/>)
- AngularJS - (<https://angularjs.org/>)
- Node.js - (<https://nodejs.org/en/>)
- Chart.js - (<http://www.chartjs.org/>)
- Angular-chart.js - (<https://github.com/jtblin/angular-chart.js>)

Appendices

A. Copyright Permissions

MySQL

MySQL Community Edition is available under the GPL (GNU Public License).

The following excerpt from GNU describes what GNU GPL is (<https://www.gnu.org/licenses/quick-guide-gplv3.html>):

Nobody should be restricted by the software they use. There are four freedoms that every user should have:

the freedom to use the software for any purpose,
the freedom to change the software to suit your needs,
the freedom to share the software with your friends and neighbors, and
the freedom to share the changes you make.
When a program offers users all of these freedoms, we call it free software.

Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

Express.js

Express.js is available for free use under the CC BY-SA 3.0 US license, which can be found at <https://creativecommons.org/licenses/by-sa/3.0/us/> and is summarized as follows:

“You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

Share Alike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.”

AngularJS

AngularJS is available for free use under the CC BY 4.0 license, which can be found at <https://creativecommons.org/licenses/by/4.0/> and is summarized as follows:

“You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material
for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.”

Node.js

Node.js is available for free use under the MIT license, which can be found at <https://raw.githubusercontent.com/nodejs/node/master/LICENSE> and reads as follows:

“Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.”

Chart.js is available for free use under the MIT license, which can be found at <https://opensource.org/licenses/MIT> and reads the same as the above.

Angular-chart.js

Angular-chart.js is available for free use under the BSD license, which can be found at <https://github.com/jtblin/angular-chart.js/blob/master/LICENSE> and reads as follows:

“Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.”

B. Datasheets

1. Thermocouple Amplifier - Maxim MAX31885 - Via Adafruit (<https://cdn-shop.adafruit.com/datasheets/MAX31855.pdf>)
2. Humidity Sensor - Texas Instruments HDC1080 - Via Texas Instruments (<http://www.ti.com/lit/ds/symlink/hdc1080.pdf>)
3. Pressure Sensor - TE Connectivity MS561101BA03-50 - Via Arrow (<http://static6.arrow.com/aropdfconversion/f23844a1f3ad2b6c222a9a9eac7ef6684c028264/31ms5611-01ba03.pdf>)
4. Triple-Axis Accelerometer - Freescale Semiconductor MMA8451 - Via Adafruit (<https://cdn-shop.adafruit.com/datasheets/MMA8451Q-1.pdf>)
5. RF Transmitter - Nordic Semiconductor nRF24L01+ - Via Nordic Semiconductor (http://www.nordicsemi.com/eng/content/download/2726/34069/file/nRF24L01P_Product_Specification_1_0.pdf)
6. Web Server - Dell PowerEdge R610 - (https://www.dell.com/downloads/global/products/pedge/en/R610_Spec_Sheet_en.pdf)