Snow Plow Mobile Data Collection and Visualization

FINAL REPORT

Team 23
Client: Henderson Products
Adviser: Dr. Goce Trajcevski
Alan Peine, Colin Heinrichs, Evan Warych, Michael Entin, Zach Wilson
sdmay18-23@iastate.edu
http://sdmay18-23.sd.ece.iastate.edu

Revised: 4/22/18 Version 2

Table of Contents

List of Figures

List of Tables

1 Introductory Material

The main goal of the project is to help with the visualization of data that is being received from sensors on a fleet of snowplows. We also discuss the environment this project will operate in, who and what this project is for and our expected end product, what our goals and stretch goals were for this semester, what our planning and development timelines were, and lastly some related products and literature.

1.1 Problem and Project Statement

Currently data is being sent from each of Henderson's trucks which contains information about the plow including its coordinates, fuel consumption, and select statistics about the plows performance. When we began this project the data was then being stored on an ftp server, but it was in an undesired format. This format made the data difficult to analyze. It existed as 1 data entry per line that needed to be input into Excel and manipulated for around an hour before it was human readable. Each type of data had to handled a different way in Excel. There was also no way to view the GPS data in any meaningful way. The Data Logger also did not perform any kind of data redundancy check or compression. This resulted in large files being sent that contain only zeros.

Ultimately the plan for the project was to create a web application that would display different statistics about the trucks that can be studied and used to make different observations. With this knowledge, quality issues can be spotted before they affect performance of the trucks, saving money from potential breakdowns. The primary use of the product would be a preventative maintenance tool.

To accomplish this our project creates a flow of data from unstructured truck logs to structured rows in a database that is queried from by our web app to visualize the data being received. At its original state the data was being stored on a server hosted by Henderson Products. Our application receives live data from the Data Logger itself, bypassing the ftp server. The product converts the data into a readable format to be stored in our database. We have designed an interface that will allow the info to be easily read and analyzed by Henderson Products' engineers and technicians.

1.2 Operational Environment

As our project deals with transferring, converting, storing, and accessing data; it is not exposed to any notable conditions with an exception for the Data Logger. The Data Logger is housed on the truck in a sealed box. Our product runs on 3 Amazon Web Service EC2 instances. These are running Red Hat Linux. These servers are capable of being accessed by employees working at Henderson most likely in an office setting. Due to the fact that it will be running on servers, it may be important to account for possible failures in that domain. Originally, since the data capture was already being captured on Henderson Products vehicles we considered the Data Loggers out of scope. This changed during the semester when we decided that it would be optimal to make changes to the software running on the data loggers.

1.3 Intended Users and Uses

The intended users of the CANBus data web application will be the employees of Henderson products and possibly their clients. Any other intended users could involve the operators of the dispensing units so that they may see the data of the vehicles they operate. It will be important to make sure that only the designated people can use this software so that Henderson's data does not get into the wrong hands.

There could be many end use cases for this app. First of all, it will reduce the need for calculations done by hand by our company contact James. This had been a very large waste of time, taking nearly an hour to convert 3 minutes of data. Key details about the data could inform the employees when things are not going right with the CANBus system. Whether that means a part on the vehicle is broken, the hydraulics readings are not what they should be, or the location is not updating. It is essential to know when things are going wrong so they can fix it as soon as possible. The data they retrieve from this app will also be used to determine better ways to create products for Henderson's clients, making their snowplows and ice work trucks able to withstand the tests of time and the harsh environment of winter.

1.4 Goals

As we began semester we had two sets of goals. There were the goals we viewed as vital to giving Henderson Products an effective tool as well as preparing ourselves for a project in the professional world. These are defined as our Base Goals below. Our Stretch Goals, also below, were things we thought would be good additional features for the project to have, or would be challenging and interesting to include.

1.4.1 Base Goals

- Efficient data conversion from hexadecimal values to relevant type
- Handling of data redundancy before being sent on network
- Easy integration for new data types
- Automatic new truck setup
- Stable database server running MongoDB
- Stable web application to display data trends
- GPS location services implemented on the website
- Secure login to website
- Scalable instances of the 3 main deliverables
- Meaningful Log data for troubleshooting

1.4.2 Stretch Goals

- Live Data from Henderson's Data Loggers
- SSL between Data Loggers and conversion service
- GPS and other data pairing
- Client-Truck pairing
- Multi-threaded conversion service

1.5 Deliverables

Our project has these main deliverables:

- Updated Data Loggers
- Data Logger Conversion Service

- MongoDB storing truck data
- Web Application
- Documentation and User Guides
- Source code for all software related deliverables

1.6 Timeline

We separated our timeline into the two semesters we have been working on our project. In this section we attempt to point out the differences between our planned timeline and our actual execution of the project.

1.6.1 First Semester

Our first semester was spent almost entirely on planning our design and doing research on the best ways to implement our project. We read through a lot of papers our faculty advisor thought may be relevant to our project, along with reading up on the documentations of the technologies we had chosen to implement in our system.

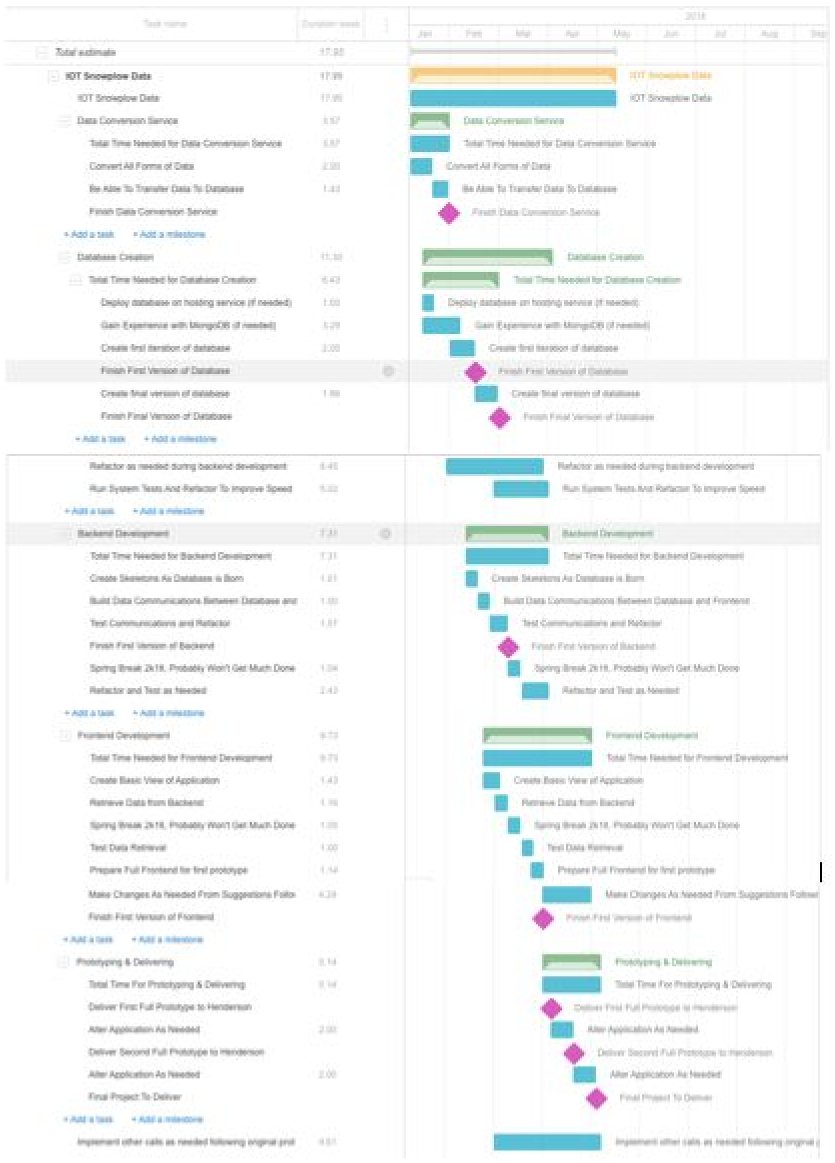
Along with design and planning, we spent a lot of time writing up our documentation for said planning and really making sure we had a grasp on the project. A lot of our first meeting with our client, Henderson, revolved around the sharing of information. We really wanted to have a firm grasp on what they expected from us as a team and the kind of things they wanted to see in the final product.

Our final goal of the first semester was to create a prototype of how the data conversion service would work and get a feel for some of the technologies it would use. We knew creating a full prototype of our project would be a little out of the scope of the semester. Thus, we decided to use Apache Thrift to demonstrate we could communicate between the data loggers and the conversion service in order to send data from the loggers to be converted.

1.6.2 Second Semester

As you can see in **Fig. 1**, we had planned to separate the work on our project based on the different components our project contains: the Data Conversion service, followed then by the database, then the backend or API, and finally the website itself. For the most part, we followed the basic idea of that timeline very closely and on time, but many of the smaller details or ideas in our timeline were either discarded or unneeded. For each section, we had the section lead headline most of the work, pulling in others when wanted or needed.

One large hiccup that occurred through the timeline this semester was working with GPS data. All other data formats we were able to keep on track and accomplish on time based on the description of our timeline. However, GPS turned out a bit of a pain to implement on the Data Conversion Service. In order to combat this, we decided to push ahead the implementation of the rest of the project in order to at least get the parts done that we could. Finally, once we had the GPS working correctly, implementing it into the rest of the project was fairly simple since we had already been creating APIs and working with the frontend technologies.



Task name	Duration week
Total estimate	17.22
IOT Snowplow Data	17.35
IOT Snowplow Data	17.35
Data Conversion Service	3.57
Total Time Needed for Data Conversion Service	3.57
Convert All Forms of Data	2.00
Be Able To Transfer Data To Database	1.43
Finish Data Conversion Service	
+ Add a task + Add a milestone	
Database Creation	11.30
Total Time Needed for Database Creation	6.43
Deploy database on hosting service (if needed)	1.00
Gain Experience with MongoDB (if needed)	3.29
Create first iteration of database	2.00
Finish First Version of Database	
Create final version of database	1.86
Finish Final Version of Database	
+ Add a task + Add a milestone	
Refactor as needed during backend development	6.45
Run System Tests And Refactor To Improve Speed	5.02
+ Add a task + Add a milestone	
Backend Development	7.31
Total Time Needed for Backend Development	7.31
Create Skeletons As Database is Born	1.01
Build Data Communications Between Database and	1.00
Test Communications and Refactor	1.57
Finish First Version of Backend	
Spring Break 2k18, Probably Won't Get Much Done	1.04
Refactor and Test as Needed	2.43
+ Add a task + Add a milestone	
Frontend Development	8.73
Total Time Needed for Frontend Development	8.73
Create Basic View of Application	1.43
Retrieve Data from Backend	1.16
Spring Break 2k18, Probably Won't Get Much Done	1.00
Test Data Retrieval	1.00
Prepare Full Frontend for first prototype	1.14
Make Changes As Needed From Suggestions Follo	4.29
Finish First Version of Frontend	
+ Add a task + Add a milestone	
Prototyping & Delivering	5.14
Total Time For Prototyping & Delivering	5.14
Deliver First Full Prototype to Henderson	
Alter Application As Needed	2.00
Deliver Second Full Prototype to Henderson	
Alter Application As Needed	2.00
Final Project To Deliver	
+ Add a task + Add a milestone	
Implement other calls as needed following original prot	9.57

Fig. 1: Our planned timeline for the second semester of our project. The second semester focused primarily on implementation of the project.

1.7 Related Products and Literature

With a new project it is important to research established products on the market that function similarly to our goals. Looking at similar products allowed our team to learn from their design decisions to figure out what kind of technological issues could potentially arise during our development phase.

One of the main drivers of using big data to gain insights into the optimization and tracking of trucks is US Xpress Transportation. While their product focuses on the performance of their freight trucks, there is still plenty to be observed by their work. Information on this can be found on their website [2] but a more concise write up on their work can be found on a technology news site called datafloq [3].

A few key issues differentiate our product from the one for US Xpress, the main distinction is their real time tracking system versus our projected system of post analysis. However, it was beneficial to learn how they use several different data sources and combined them for analytics, this is a key feature of our application. Furthermore, getting familiar with the idea of geospatial analysis was helpful as we do plan to map out the trucks data in a similar fashion.

During our project the Iowa Department of Transportation released “Track a Plow”[1]. This web application shows live data from snow plows, traffic cams, and weather devices. It allows a user to see what the conditions of the roads are. There is color-coded paths of the snow plow to indicate how passable this road is. This is similar to one of our earlier prototypes that paired information like salt distribution or engine speed to GPS coordinates.

2 Design and Implementation Details

2.1 Proposed Design

We designed a fairly simple data flow solution. Our general overview of the components is shown below in **Fig. 2.** We designed a Java service that could receive data directly from the loggers and convert it from hexadecimal bytes to its relevant size and type. This data was then sent to the MongoDB database. When a user wants to view the spreader rate data from the day before they can set those parameters on the Javascript frontend. A call is made by the Java backend to the database to deliver for displaying. A full system block diagram can be seen in **Fig. 12** in the appendices for Design Materials (5.2).

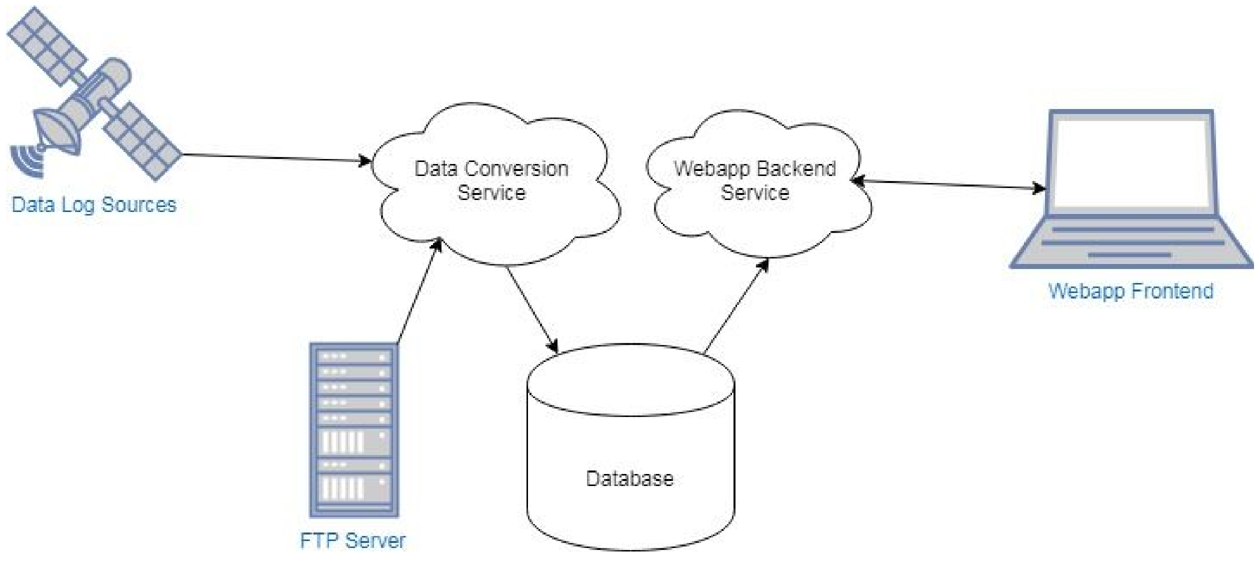


Fig. 2: Design diagram describing the proposed layout of the different parts of our project

2.1.1 Functional Requirements

- **Data Conversion** - Currently data is being hosted on an ftp site in the form of obfuscated log files filled with hexadecimal values. This data needs to be converted into something that makes more sense for humans.
- **Data Storage** - The converted data will need to be stored somewhere where it can be queried for in useful ways for analysis.
- **Data Access** - An interface will need to be build that gives an intuitive way to query for data that will be useful.
- **Data Analysis** - An interface will be needed to outline useful trends in data using graphing tools.

2.1.2 Non-Functional Requirements

- **Security Requirements** - Our project should limit access to data to the people that need it within Henderson products. This could mean locking it down to an internal network, or requiring some sort of authorization to access, or some combination of the two.
- **Responsive Requirements** - Since our project is a web app, our applications should be fast enough to meet the needs of a company. Any request for data should take no longer than 5 seconds to complete and populate on the page.
- **Time** - This project will need to be working and have most of the core functionality before May 2018.
- **Cost** - We will need to consider costs when determining the overall architecture of our project, as this will affect the cost concerning server usage.
- **Software Licenses** - Because our project will be used in a commercial setting, we will need to be sure that the licenses for any of the software libraries we are using gives us rights to use it for this project.
- **Vehicle Access** - We will have limited access to the trucks that the data is being logged from. This should not have any major effects on our project, as the data transfer is already being handled, though having access to the trucks may give us a better idea of what the data is for.

2.2 Data Loggers

The Data Loggers consist of a Raspberry Pi, a PiCan2, a PiRTC, a DC-DC converter, a battery, and a Sierra Wireless Airlink. The PiCan2 allows the Pi to connect to the CAN network. The PiRTC gives the real time to the Pi. The Pi connects to the internet and thus our Data Logger Conversion Service through the Airlink. The DC-DC converter steps the vehicle 12v down to 6v for the Pi. It also allows the Pi to keep until it has finished transmitting data, even if the truck has turned off. The battery assists with this. The Pi is utilizing an opensource can-utils library to dump all CAN messages. Several bash scripts work together to take this data into a file and push that file to the ftp server. During our initial planning we decided we would just supply a filter to this process to remove the redundant data and continuing to output batch files, which would be uploaded later by a python client for our Data Log Conversion Service. This turned out not to be the case.

We ended up writing our client for the data loggers in python, which monitors the CAN network and sends data records to the Data Logger Conversion Service. One of the biggest issues that we wanted the data logger client to solve was the issue of limiting the amount of data that is being sent to be stored, as in many cases identical messages on the CAN network were being logged several times per second. This redundant data is detrimental to our project, because it does not help us make useful observations, and it takes up extra space in our database and bandwidth on the mobile networks being used by the trucks.

2.2.1 Transition to Real-Time Data Streaming

Our final implementation of the data logger code monitors the CAN network and sends data records to the Data Logger Conversion Service periodically in batches of data records. The size of these batches are dependent on two values which are configurable through the Data Log Conversion Service, and are WINDOW_BATCH_SIZE, which is the number of data records needed to trigger an upload; and WINDOW_BATCH_TIMEOUT, which is the number of seconds since the last upload that need to pass before an upload is triggered regardless of the number of records that have been accumulated.

2.2.2 Log Filtering

We decided that it was best to be able to configure how logs are filtered based on the COBID of the message, which corresponds to the type of the CAN message. This configuration gives a policy for each COBID, and is given to the data loggers by DLCS on startup. These policies can currently be set to one of the following:

1. Always log data records with this cobid
2. Never log data records with this cobid
3. Only log data records with this cobid if they contain values different from the last values logged

If no configuration is set for a particular cobid, the data loggers will default to the always log policy. The third policy is our main attempt at a solution for limiting logs, and works by caching the last log sent for a particular cobid and comparing it to new ones to see if the byte payloads are different. Only logs with different byte payloads are sent to be published. The cache used to hold the last log for the different cobids only persists for the duration of a batch of records (see **section 2.2.1**), after which the cache is emptied. This means we effectively create a tumbling window over which this filtering takes place, guaranteeing that we will publish at least one log for each cobid observed within the window, which is illustrated in **figure 3.**

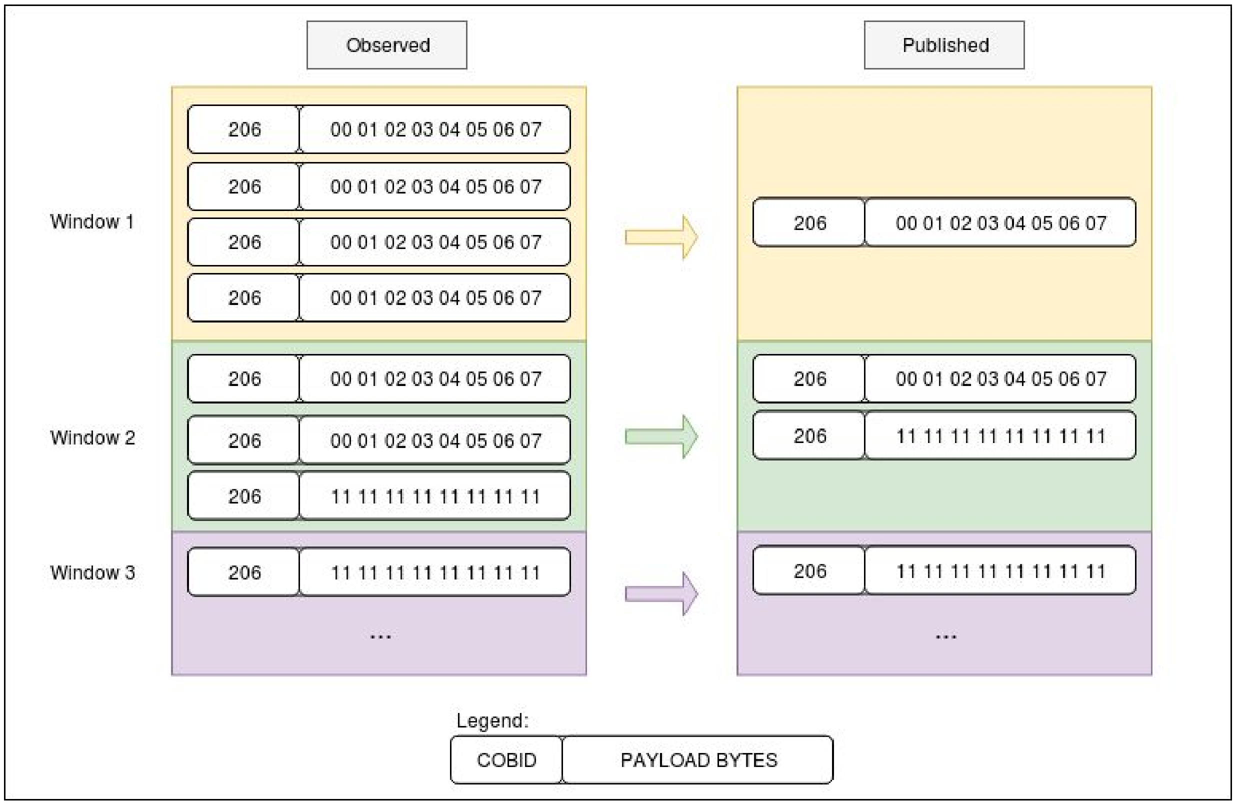


Fig. 3: Windowed Filtering of Unchanged Values

The figure shows the data logger client's view of the logs with COBID '206' in a stream of data records, and the logs that are left after the filtering process. As you can see it filters out duplicate records within each window, while still guaranteeing that at least one record for an observed cobid will be present within that publishing window, making it a pretty safe method for reducing the number of data records sent.

2.2.3 Fault Tolerance

There are a lot of things that can go wrong while a data logger is in operation, and we realized that it is really important to write the data logger client to handle these issues in sensible ways so that they do not cause our client to break. The main issues we identified are as follows:

1. Initial connection issues with the Data Log Conversion Service
2. Intermittent lost connections during operation
3. Missing config values from Data Log Conversion Service

We combated the first issue by simply retrying to connect to the Data Log Conversion Service every 5 seconds if we fail to connect, as in this case it is likely that the access point has not had time to connect to the mobile network yet. We realized that it is possible for us to lose our mobile connection mid-operation, and lose our ability to publish logs. In this case we save the batch of logs as a file with a timestamp in a configurable location to be published later by a separate script that runs periodically. Finally, we decided we needed to be weary of missing configuration values, and to have sensible fallbacks wherever possible so that we can continue operation.

2.3 Data Logger Conversion Service

The Data Logger Conversion Service (DLCS) needed to be carefully designed and implemented, as it would be handling a lot data. When we began this project there were hundreds of data files stored on Henderson's ftp server. We needed DLCS to be as efficient as possible. We wanted to make sure that converting the data would not be the bottleneck for the system. DLCS would need to handle the first packet sent from a new truck, a trucks first appearance of a COBID, or even how to handle unconfigured COBIDs. DLCS needed to be up and running as fast as possible to help provide assurance and design stability to the elements later on in the data flow. Lastly, for reference a full Domain Model of DLCS can be seen in Appendices 5.2 **Fig. 13**.

We first set up a solution for communicating with Henderson's Data Loggers. We were not initially sure what we would need to do on the Data Logger side. This is one of the things that lead us to using Apache Thrift. Using thrift forced us to clearly define our interfaces and models. It also allowed us to abstract the client communication away. Apache auto-generated the data class DataRecord for us. We defined a LogServiceModule and a LogServiceHandler. When the server is started it uses Google's Guice framework to inject the dependencies defined in LogServiceModule into our LogServiceHandler. The dependencies in the LogServiceModule are things like the database connection, which handler to use, and then eventually the service's COBIDReader.

With the incoming communication handled we started focusing with how we were going to convert the data. We knew that if Henderson ever added to their product line, they would need to be able to parse new COBIDs. The existing COBIDs also vary in terms of signed and unsigned, the type of data, as well as the size. We needed to do byte manipulation and handlings of whether or not a byte is signed, in a language that does not have unsigned bytes. Therefore we needed a solution that could change how its parsing data with each piece of data as well as makeup for some faults in our chosen language. It also had to be able to easily be updated with new translations. This lead to the creation of COBIDReader. COBIDReader's primary function comes from a json configuration file. A sample part of this file can be seen in Appendices 5.2 : **Fig. 14**. Each COBID has a unique integer identifier in this case "206". There are 5 array elements inside 206. This means that there are 5 pieces of information sent in this packet. Each element has a description of the type of data, the amount of bytes it uses, the type of data, and a division adjustment value. Whenever a value needs a decimal it must first be parsed into a double and then divided by the adjustment. For each element in this array, COBIDReader steps through the dataLength in the list of bytes. Once parsed, they are stored as key-value pairs in a Document, and the next translation is accessed. All of the COBID translations are stored in a HashMap<int, array> for maximum efficiency and convenience.

With the data converted it was time to send it to the database. LogServiceHandler is what handled this. If it received a single data record it added a single new piece of data into the database. This is really inefficient for parsing lots of data so we also have added a way to add a large collection of data at once efficiently. When the LogServiceHandler first starts up it queries the database for the trucks currently stored in it. When data passes through the handler we check its TruckID against our cache of trucks. If it does not appear the handler will update the database. The same things happens when a translatable COBID passes through for a truck for the first time. This allows Henderson to view data from trucks as soon as it passes through the service.

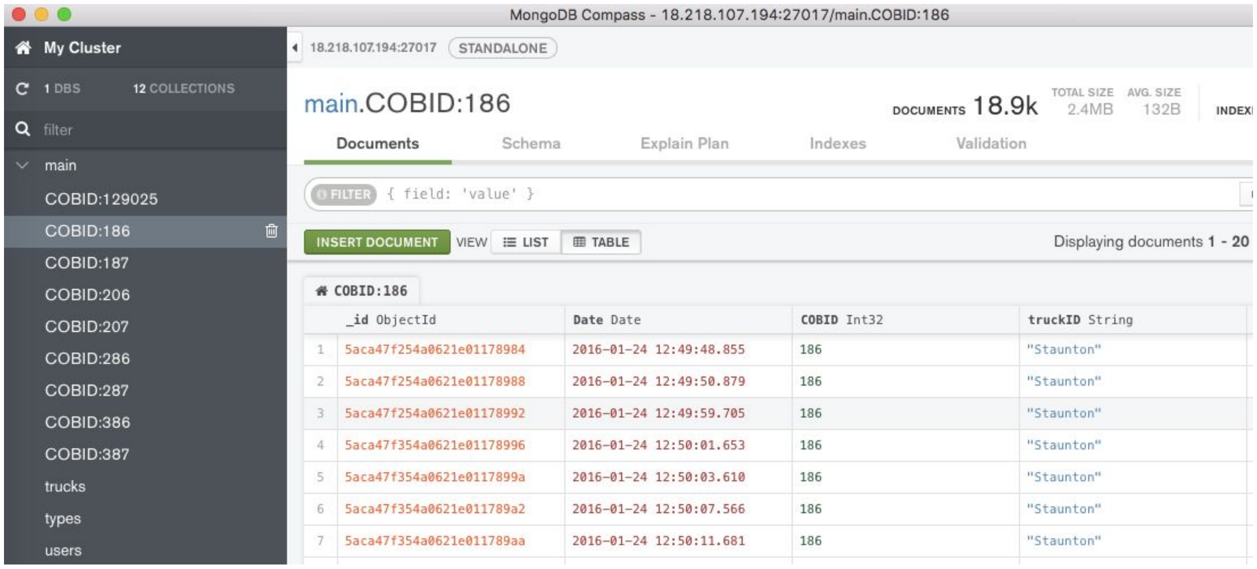
The last part of DLCS was created because of the large amount of data in the ftp server. We first prototyped DataParser during our first semester. We had to do some efficiency upgrades as well as format changes. It was built as a file parser in our early designs. Once updated we created a script that could be run once. It would comb through the ftp server, locate all of the old log data, attach its related identification parameters, and send it through DLCS as if it had received it from a Data Logger.

2.4 Database

The database we chose to store the data converted from the DLCS is MongoDB. This was chosen as we are dealing with large amounts of data coming in every minute and the horizontal scalability that is provided by MongoDB is ideal for this situation. The instance of the database itself is running on an AWS EC2 instance provided by amazon. This design decision per Henderson's request, and setting up the AWS instance with MongoDB was quite a challenge.

As we were deciding on using MongoDB, we had checked out all the documentation on implementing it with different technologies and it seemed like it would work well on any platform. Then, as we attempted to use it in Amazon's EC2 instance, we could not get it working. It looked like we might have to choose a different option for either the database or where we were hosting it. Then, we decided to look into what type of system the EC2 was running on, and we discovered it was a Red Hat SELinux machine. In Mongo's documentation, it describes that there is one configuration change needed to allow it to start on those systems. After making this change, everything was working smoothly.

The actual database itself has many collections for storing each COBID's data. As is visible in **Fig. 4**, each of these has a truckID to associate with which truck was logging said data. We then have collections for storing which COBIDs are associated with certain names, users for authentication, and the trucks themselves. All of the COBID collections are also indexed by truckID in order to further decrease query time.


MongoDB Compass - 18.218.107.194:27017/main.COBID:186

My Cluster
1 DBS 12 COLLECTIONS
main
COBID:129025
COBID:186
COBID:187
COBID:206
COBID:207
COBID:286
COBID:287
COBID:386
COBID:387
trucks
types
users

18.218.107.194:27017 STANDALONE

main.COBID:186

DOCUMENTS 18.9k TOTAL SIZE 2.4MB AVG. SIZE 132B

FILTER { field: 'value' }

Displaying documents 1 - 20

COBID:186

	_id ObjectId	Date Date	COBID Int32	truckID String
1	5aca47f254a0621e01178984	2016-01-24 12:49:48.855	186	"Staunton"
2	5aca47f254a0621e01178988	2016-01-24 12:49:50.879	186	"Staunton"
3	5aca47f254a0621e01178992	2016-01-24 12:49:59.705	186	"Staunton"
4	5aca47f354a0621e01178996	2016-01-24 12:50:01.653	186	"Staunton"
5	5aca47f354a0621e0117899a	2016-01-24 12:50:03.610	186	"Staunton"
6	5aca47f354a0621e011789a2	2016-01-24 12:50:07.566	186	"Staunton"
7	5aca47f354a0621e011789aa	2016-01-24 12:50:11.681	186	"Staunton"

Fig. 4: MongoDB's community edition GUI MongoDB Compass displaying all of the collections (left side under main) and some fields that are common throughout the COBID collections (Date, COBID, truckID).

2.5 Webapp Backend

Our Webapp API was implemented using the Java Spring Boot framework and was also deployed on an Amazon AWS EC2 instance, which we worked with Henderson Products to have set up and configured. We chose to use Spring Boot in our application in order to have a RESTful API in a language we are all familiar with. It also decreases the amount of time spent on having to "reinvent the wheel" when it comes to making and designing a RESTful service.

There were a lot of steps in creating our API, the first of which was setting up the Spring Boot project and giving it continuous integration capabilities on our git repository for the project. Next, we needed to find a dependency for using the MongoDB API within our system. This took quite a bit of time to figure out, as the most up to date system of interacting with MongoDB is currently incompatible with Spring Boot. This caused us to use Spring Boot's MongoDB starter pack, which was less than desirable, as the classes used to query data were documented worse than the current version.

Finally, it came down to designing all of the different controller classes and actually creating all of the API calls. First, we had to set up a MongoConfig file in order to connect to our database when making said calls. Then, we created a class named MongoInteraction which contains a protected autowired variable which contains the connection to the database. Then, all of our Controller classes need only extend that class in order to connect to the database. We have three controllers on our API: one for user endpoints, one for all of the data associated with the mechanisms running on the truck, and a final one for GPS data endpoints. These are named UserController, TruckController, and GPSController, respectively. We felt like GPS and the rest of the data associated with the truck needed to be separated because they are displayed differently on the webapp. Thus, clumping them together might cause the application to slow down since it would have to wait for both types of data to arrive before being able to populate either of the two ways data can be shown to the user.

2.6 Webapp Frontend

Our Webapp frontend is a single-page web application that is developed using the EmberJS framework. We decided to build a single page web application for multiple reasons. First, we did not want the users of the application to have to install our application on their machines. Instead, they just simply have to open up a modern web browser (an application that they already have installed) and navigate to the appropriate page. Second, we wanted to separate the data from the views. This means that the client can get raw access to the data without using our web app.

We chose EmberJS over other frameworks, such as AngularJS. EmberJS is slightly faster, especially when dealing with large amounts of DOM updates. Our application could have a lot of these updates especially when users want to modify any sort of data on a graph (this could include colors, which data is being plotted, etc.). Lastly, we liked Ember's best practices and syntax more than Angular. We believed that Ember will make developing our single page application easier.

Additional libraries that we used to develop the frontend included: ChartJS, Google Maps, and EmberPaper. ChartJS is an open-source graphing library written in javascript. ChartJS allows us to display the truck data as a function of time to the users. Google Maps is used to display the most recent location of the trucks to the user. Additionally, we use Google Maps to draw paths that a truck has taken. Finally, EmberPaper is a library that incorporates material design principles into EmberJS. This is used to provide the overall theme and structure of the application.

3 Testing Process and Results

In this section we outline our testing plan throughout the project and discuss each part of it. We cover the results of our testing as well as a critique of ourselves as it pertains to the testing of our project.

3.1 Testing Plan

Our initial testing plan that we came up with in the Fall semester consisted of four stages, as well as conditions required for transition between them. As shown in **Fig. 5**, these stages are: Development, Review and Merge, PreProd, and Prod.

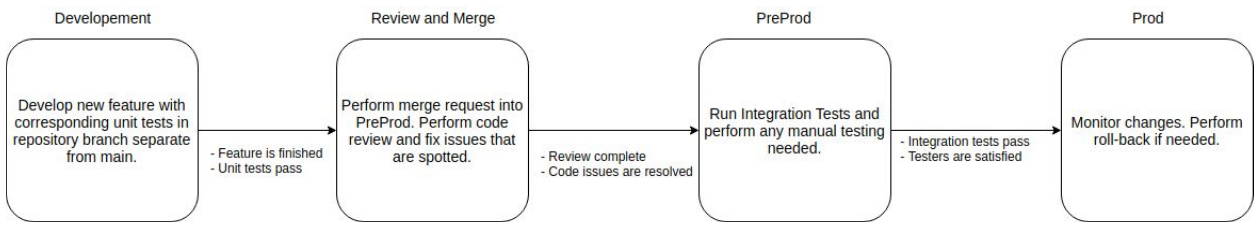

Developement
Develop new feature with corresponding unit tests in repository branch separate from main.
- Feature is finished
- Unit tests pass
Review and Merge
Perform merge request into PreProd. Perform code review and fix issues that are spotted.
- Review complete
- Code issues are resolved
PreProd
Run Integration Tests and perform any manual testing needed.
- Integration tests pass
- Testers are satisfied
Prod
Monitor changes. Perform roll-back if needed.

Fig.5 - Initial testing plan

For the most part this testing plan was left unchanged for the Spring semester. The main difference is the exclusion of the prod stage as originally defined. Our team decided that it did not make sense to maintain a production stage while much of our project was in development; as it would not get used until our project reached the appropriate maturity. Because of this our team decided to leave the production stage out to save costs for Henderson Products we decided to leave the production stage out. At our current state we believe our project is production ready, meaning it is ready for real-world use. Keeping this in mind we also strongly feel that it is necessary to keep a separate environment for testing away from the production environment. This means that any future work on our project should be tested on a similar but separate environment to what our project is currently running on.

3.2 Unit Testing

Our earliest method for catching defects was to perform unit tests for the different parts of our application. These are meant to ensure that the individual units of functionality in our project work as expected, and also act as regression tests by ensuring that future changes do not break current features. These tests were written using popular frameworks for unit testing, namely JUnit for our Java applications, and PyUnit for the python code written for the data loggers.

Our use of Gitlab's continuous integration features allowed us to run these tests automatically after every commit and to view the results. This made it easy for us to know when things are broken so that we can recognize and fix the issues. It also keeps us from advancing along the testing pipeline while things are still broken.

3.3 Integration Testing

Our integration tests are meant to ensure that the different parts of our system work together correctly. These tests are manual for the most part, and are performed in our "preprod" staging environment. It is usually pretty apparent when things are not interfacing with each other correctly, as our project was designed to fail quickly in such cases.

3.4 Quality Assurance

As stated in our initial testing plan, we decided to perform code reviews in the "Review and Merge" stage. This stage is initiated when a developer creates a merge request into master for their work, at which point another developer is expected to review the changes using gitlab's interface before merging them into the master branch of our repository. If issues are spotted by the reviewer they are expected to notify the creator of the merge request, and these issues are expected to be resolved before the merge takes place. This allows us to spot and fix code quality issues before they are allowed to persist in the master branch of our repository.

3.5 Testing Results

Overall we feel our team did a good job of following our testing plan, but we also feel we could have adhered to it more than we did in several areas. One of the big areas we feel we could have done better on is our unit testing, which we feel we started strong on at the beginning of the semester, but also saw test code additions decline as the semester progressed. One of our biggest successes for testing was our testing of the log conversion functionality, which we feel we did a great job of writing unit tests for every COBID we had configured for conversion. Because of this we are very confident in DLCS's conversion functionality.

Our integration tests were adequate for catching most issues early, and our setup allowed us to perform the tests in a production-like environment, so we are very confident that our systems will perform correctly. We would have liked to automate more of the integration tests, but had to make tradeoffs due to time constraints.

Our code reviews worked well for spotting issues, as well as for encouraging our teammates to look at each others' work and learn about different parts of the project. As the semester progressed however we saw code reviews becoming less and less involved, with less comments being made during review.

4 Closing Materials

In this section we review and conclude on our plan for this project. We also acknowledge and thank people that have helped us along the way.

4.1 Conclusion

After a long year of gathering requirements, researching, working with Henderson, working with AWS, developing and testing code, and everything else in between, we are proud to say we have created a web application we can be proud of. We have successfully created a tool which Henderson can use to convert and visualize all of their snowplow data. We have had some struggles and some setbacks, but in the end we completed all of the goals we set out to accomplish. We also managed to finish our first stretch goal of sending live data.

4.2 Acknowledgment

Thank you to James Timmerman, Shane Chesmore and Henderson Products for working with us and providing us with the information for our project. Thank you, Dr. Goce Trajcevski, for your help and guidance with the the direction and management of our project. As well as encouraging us to submit our demo-paper to the IEEE Mobile Data Management in Aalborg, Denmark.

5 Appendices

We have 2 appendices. The first one serves as a operations manual that goes into how to setup, configure, and use each part of the project that could need reconfiguring or instructions. The next is a reference of design materials mentioned throughout the project.

5.1 Operation Manual

This appendix serves as a guide on how to setup, configure, maintain, scale, or use the Data Loggers, Data Logger Conversion Service, and the Web Application.

5.1.1 Data Logger

Our modifications made to the data loggers consist of several python files present in the data-logger-client folder of our main repository. There are two main scripts: 'publishFromStdIn.py' and 'publishFailedRecords.py', which both depend on other modules in the project. The publishFromStdIn.py script is meant to monitor the stream of logs that are being generated by the Candump utility through stdin, e.g. by running candump and piping its output to 'publishFromStdIn". The 'publishFailedRecords' script is meant to check a specified folder for logs that failed to send earlier, in which case it uploads all of the output of any files it finds, and afterwards deletes the file.

5.1.1.1 Setup and Deployment

To make deployment easier we decided to package our two scripts using "pex", which is a tool for packaging python programs and their dependencies into a single file, which can be executed directly on any machine with the correct version of python installed. To make things easier a Makefile is provided that builds these automatically, but requires Apache Thrift and Python Pex to be installed in the machine it is being built on which, because of the way pex builds work, will need to be a raspberry pi or similar device with an armhf processor. Once these conditions are met, the "make" command can be ran from the data-logger-client folder, which will create two .pex files in the build directory of the data-logger-client folder.

Once created these .pex files will need to be copied onto a usb drive attached to the raspberry pi data logger, along with a config.ini file (**see section 5.1.1.2**) and the modified 'start_logger.sh' file (present in the "data-logger-client" folder). The 'start_logger.sh' file will need to be configured to run when the data-logger starts up using the Cron System tool. The 'publishFailedRecords.pex' file will also need to be configured to run periodically as a Cron job.

5.1.1.2 Configuration

Both 'publishFailedRecords.pex' and 'publishFromStdIn.pex' take a path as there a command line argument, which leads to a .ini configuration file that contains values necessary for the scripts to function. These values are:

```
TRUCK_ID                 # The id of the truck the data logger is on.
DLCS_HOSTNAME            # The address of DLCS
DLCS_PORT                # The port DLCS is running on
FAILED_LOG_PUBLISH_DIR #Where to put/find failed records
```

A version of this file is in the ‘data-logger-client’ folder of our main repository, which can be used as a reference for what this file should look like.

5.1.2 Data Logger Conversion Service

We designed DLCS to be easy to setup, configure, maintain, and scale. In this section you will find manuals to start DLCS, configure it, check its log data, and potential scaling.

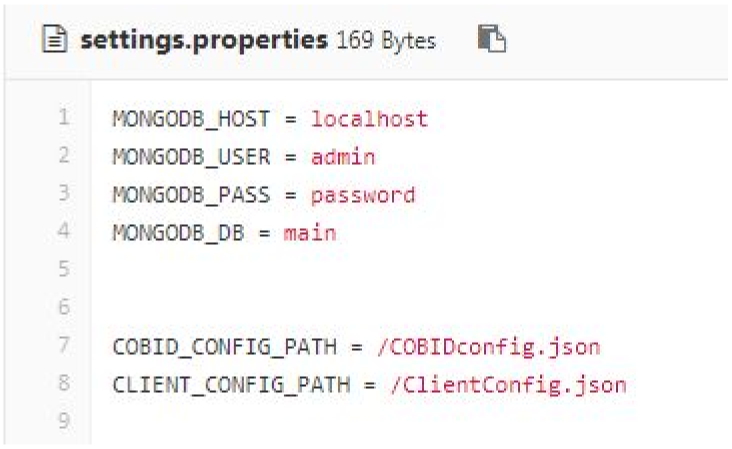
5.1.2.1 Setup

To start DLCS you will need to access the AWS EC2 instance associated with the service. You will need your credentials in order to do this. We have setup continuous integration for DLCS. Whenever you push code to the master branch in the repository it will run a Gradle build to set up a start script for the service. All of this will be located in data-log-service.zip. However, we have set up a Linux systemd service file. When on the instance you can simply type “systemctl start/stop/status logservice” into the terminal to either start, stop, or check if its running.

If you are deploying to a new server you will need to unzip data-log-service.zip. In this folder is a start script that is created when Gradle builds the project. Entering ./data-log-service will start DLCS as long as it has been configured. You can set up the systemd service file according to your Linux distribution instructions. This will allow you to start and setup the service as described above.

5.1.2.2 Configuration

There are 3 files that you need to edit when configuring DLCS. These are: settings.properties, COBIDconfig.json, and ClientConfig.json. COBIDconfig.json and ClientConfig.json will be included in a new deployment of DLCS. In the case of a new deployment settings.properties will need to be created inside the folder. It should look similar to **Fig. 6** below. You will need to set MONGODB_HOST, MONGODB_USER, AND MONGODB_PASS to their correct values. The client and cobid configuration paths should be the same.

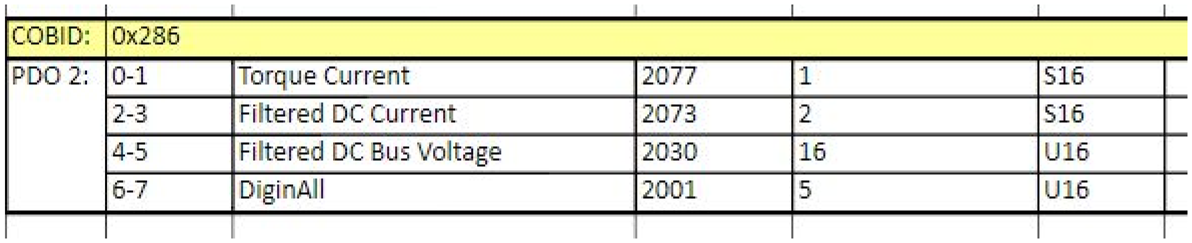


```
MONGODB_HOST = localhost
MONGODB_USER = admin
MONGODB_PASS = password
MONGODB_DB = main


COBID_CONFIG_PATH = /COBIDconfig.json
CLIENT_CONFIG_PATH = /ClientConfig.json

```

Fig. 6: Sample settings.properties configuration file.

When a Data Logger connects to DLCS it asks for its configuration policy. This is maintained inside ClientConfig.json. For more information about configuring the Data Loggers see Appendices 5.1.1.2.

You can see a sample of COBIDconfig.json in **Fig. 14** in Appenices 5.2. This file is what enables DLCS to translate the data sent by the Data Logger into a more readable format. If you wish to add translation for a new COBID you will need to edit this file. It will probably be beneficial to edit this in IDE that knows JSON syntax in order to guarantee continued functionality. You will want to go to the end of the file and find the last "]". Here in **Fig. 7** there is a sample breakdown up a COBID.



COBID:	0x286				
PDO 2:	0-1	Torque Current	2077	1	S16
	2-3	Filtered DC Current	2073	2	S16
	4-5	Filtered DC Bus Voltage	2030	16	U16
	6-7	DiginAll	2001	5	U16

Fig. 7: Sample COBID breakdown of 286.

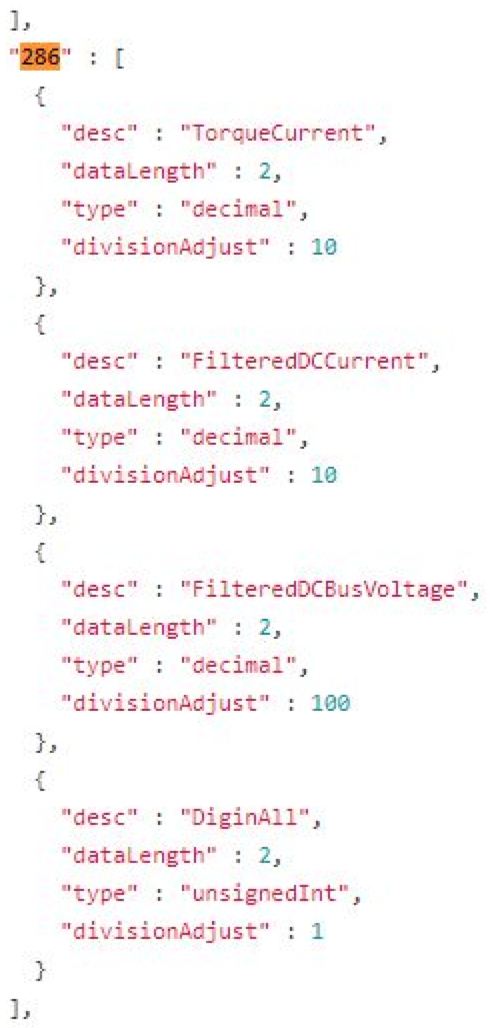
As you can see there are 4 different data values. Each 1 is using 2 bytes. The last column is telling you whether or not the value is signed or unsigned. This will matter when you set the type field. We have coded DLCS to know how to parse 6 data types as described in **Table 1.**

Type	Byte Length
ubyte	1

decimal	2
decimal4	4
bool	2
unsignedInt	2
signedInt	2

Table 1: Configured Datatypes for COBIDconfig.json

Fig. 8 shows the completed configuration of COBID 286. You will notice that the first 3 items are set as decimal. This is because they are supposed to have a decimal number for higher accuracy. The divisionAdjust is set as not “1” in order to do this. If a number needs a decimal, it must be set as either a decimal or decimal4 type. If this is the last element in the file, make sure the last “]” does not have a comma after it.



```
],
"286" : [
  {
    "desc" : "TorqueCurrent",
    "dataLength" : 2,
    "type" : "decimal",
    "divisionAdjust" : 10
  },
  {
    "desc" : "FilteredDCCurrent",
    "dataLength" : 2,
    "type" : "decimal",
    "divisionAdjust" : 10
  },
  {
    "desc" : "FilteredDCBusVoltage",
    "dataLength" : 2,
    "type" : "decimal",
    "divisionAdjust" : 100
  },
  {
    "desc" : "DiginAll",
    "dataLength" : 2,
    "type" : "unsignedInt",
    "divisionAdjust" : 1
  }
],
```

Fig. 8: Completed configuration of COBID 286.

Once DLCS is configured in this way, anytime it encounters this COBID DLCS will be able to acknowledge this is new for that truck and update the database and website accordingly. However, data it has already encountered with this COBID will not appear in the database. DLCS is not backwards compatible in this way.

The last config file that can be modified is the ClientConfig.json, which is what DLCS uses to tell the data loggers how they should operate. The file has the following format:

```
{
  "WINDOW_BATCH_SIZE": 1000,
  "WINDOW_BATCH_TIMEOUT_SECONDS": 60,

  "filterPolicies": {
    "206": {
      "policy": "ALWAYS_LOG"
    },
    "351": {
      "policy": "FILTER_UNCHANGED"
    }
  }
}
```

Fig. 9: ClientConfig.json format

Here you can change the values that will be used to determine the size of the batches that data loggers will accumulate before publishing by changing the "WINDOW_BATCH_SIZE" and "WINDOW_BATCH_TIMEOUT_SECONDS" values, and can also add filter policies for different COBIDs, which the data loggers will use to determine how to filter the logs it observes (See section 2.2). This file is in the 'src/main/resources' folder of the DLCS project directory. It will be included in the jar file generated by 'gradle build' automatically, and no additional steps need to be taken to include it in the deployment.

5.1.2.3 Maintenance

Whenever DLCS encounters a COBID it does not know how to translate, or data in a format it is prepared for it will log it. You can view this log by checking DataLoggerConversionServiceLog.txt. This file will contain when and why it encountered an error. The log appears in chronological order from top to bottom.

If a COBID you want translated is being sent but not translated, this will be the file to check. It is also where you could check if you are having issues with a newly configured COBID.

5.1.2.4 Scaling

DLCS is meant to be scalable. It cannot currently be configured to increase its size. JavaServe has a section in it that meant to be where multithreading could be implemented. Adding code there would increase DLCS's potential.

5.1.3 Web Application

The Webapp Frontend is designed to be easy to deploy and use. The following setup instructions are designed for deploying on AWS, however, this application can be hosted on any server that can serve static files.

5.1.3.1 Setup

In order to develop and build the application make sure that you have NodeJS and EmberJS installed on your computer.

Next, run the following command in the root directory of the project: **npm install**.

There are two values that might need to be modified depending on both the location of the backend and if you want to use your own Google Application Key. These values can be found in **config/environment.js**. Makes sure that the value for **backend-url** is set to the url for the root of the backend API (This only needs to be done if the backend is ever moved).

To serve locally, run the command: **ember serve**

To build the application, run the command: **ember build**. You can then deploy the contents of the dist folder to where you want the application to be hosted.

5.1.3.2 User's Guide

The application should be easy and straightforward to use. The following are some useful functionalities of the application.

The two base pages of the application are the index page and trucks page. These are located at "/" and "/trucks" respectively. The index page (see **Fig. 10**) displays the last updated GPS locations of all of the trucks in the system. The trucks page lists all of the trucks and provides links to data and track truck.

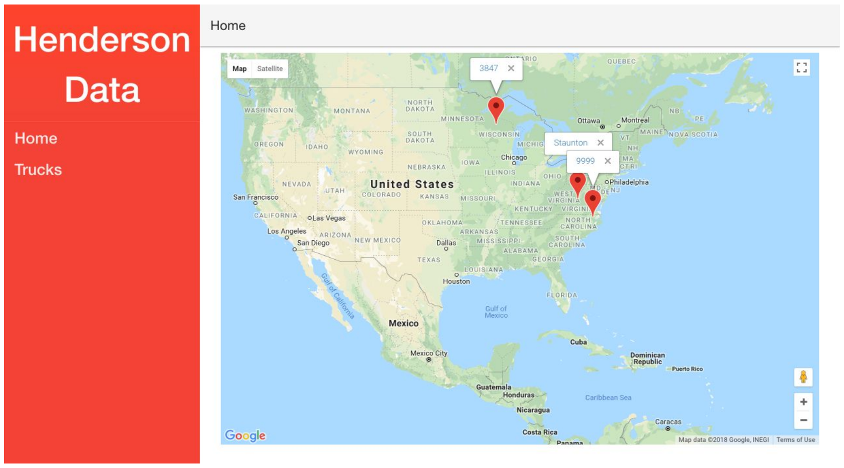


Fig. 10: Home page of the webapp.

Clicking on data brings up a list of COBIDS and specific data points. Clicking on a COBID will take you to a page where you can view all of the data for a specific COBID and export that data to a CSV file. Clicking on a specific data type bring up a graph for that data type (see **Fig. 11**). Here, you can specify a date range that you would like to see data from and it will display in on the graph. Double click the graph to put it into full screen mode.

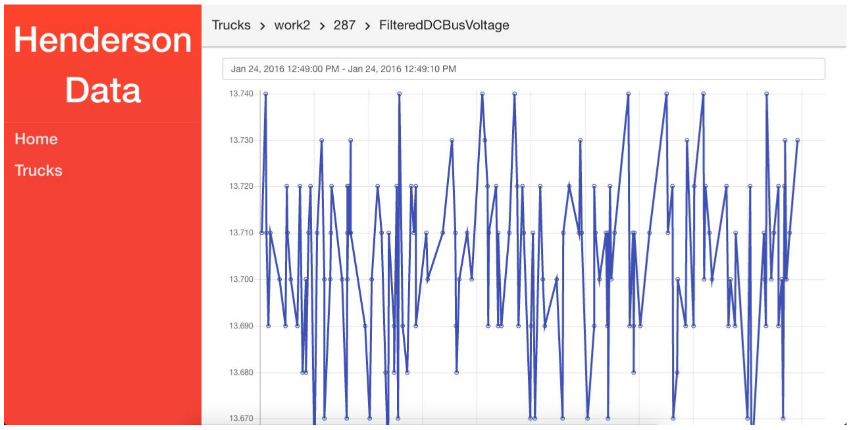

Henderson Data
Home
Trucks
Trucks > work2 > 287 > FilteredDCBusVoltage
Jan 24, 2016 12:49:00 PM - Jan 24, 2016 12:49:10 PM
13.740
13.730
13.720
13.710
13.700
13.690
13.680
13.670

Fig. 11: Example graph of a data type with date-range selector

Clicking on track truck will bring up a Google Map that displays the path that a truck took over a given time period. You can specify the time period at the top of the page and the map should display the GPS data accordingly.

5.2 Design Materials

In this Appendix we present several design materials relevant to what we have developed.

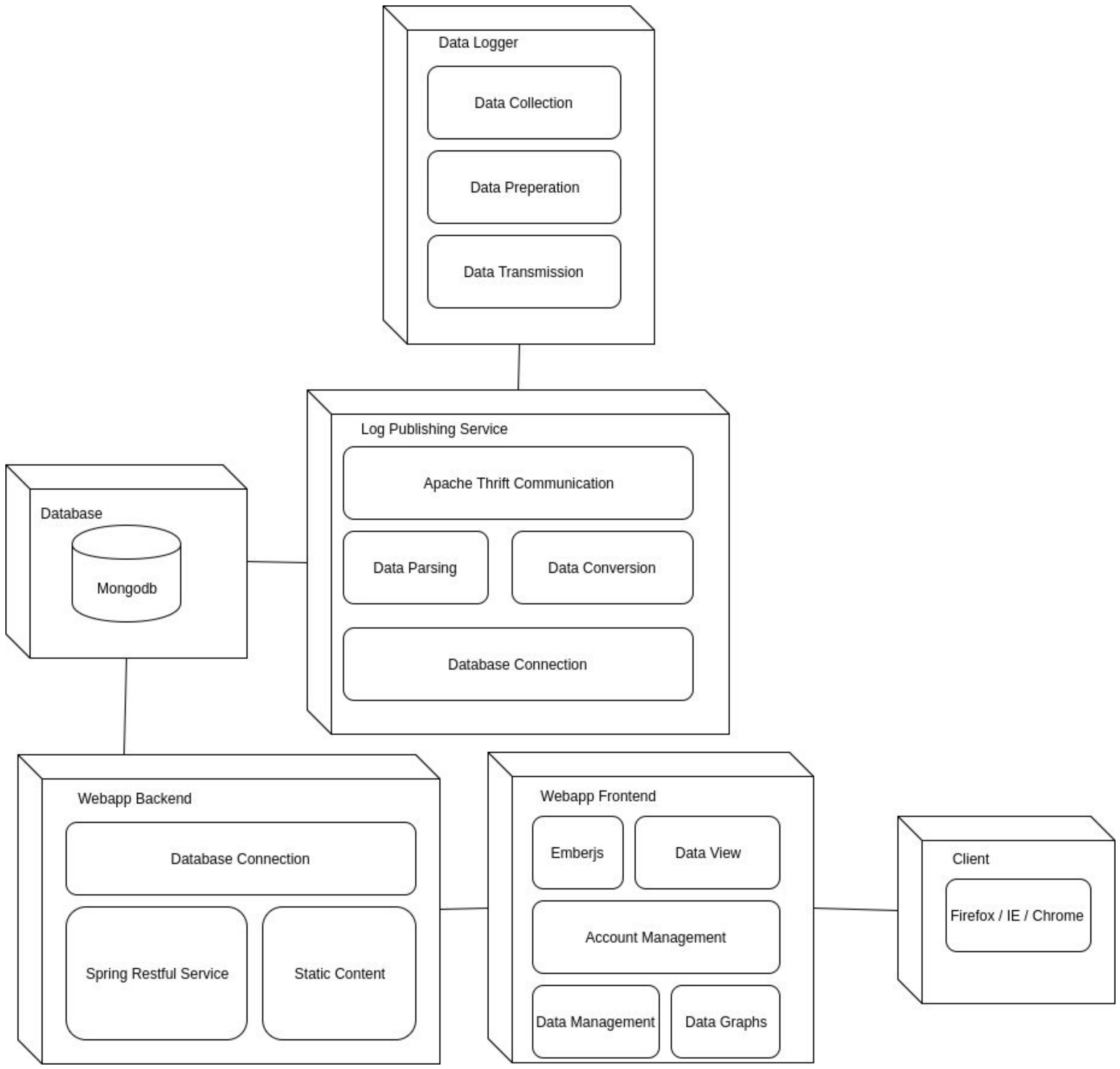


Fig. 12: Proposed System Block Diagram

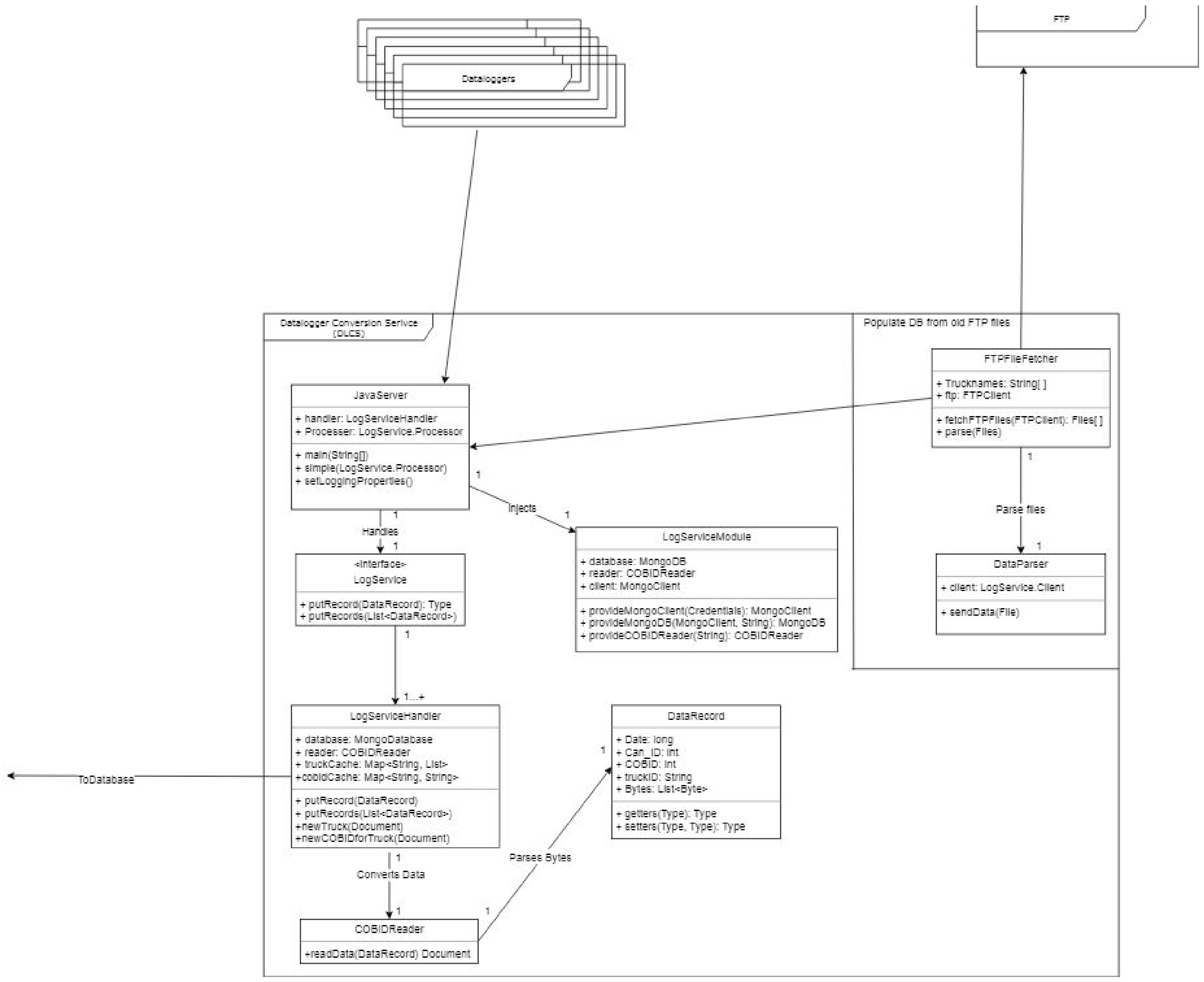
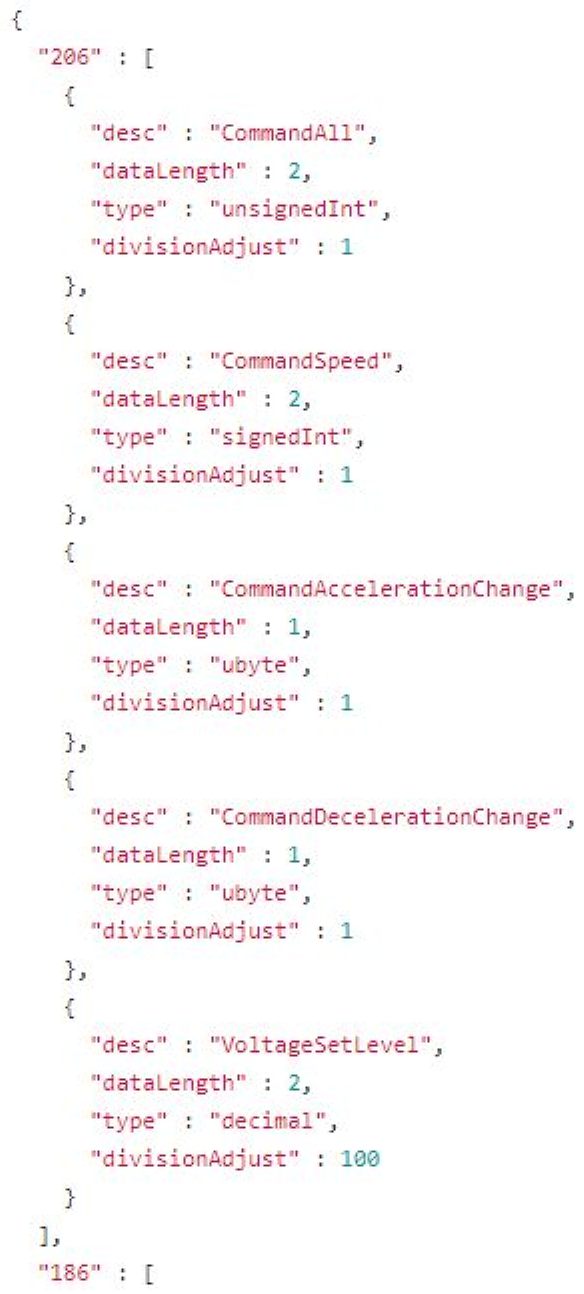

FTP
Dataloggers
Datalogger Conversion Serivce (DLCS)
Populate DB from old FTP files
FTPFileFetcher
+ Trucknames: String[ ]
+ ftp: FTPClient
+ fetchFTPFiles(FTPClient): Files[ ]
+ parse(Files)
JavaServer
+ handler: LogServiceHandler
+ Processer: LogService.Processor
+ main(String[])
+ simple(LogService.Processor)
+ setLoggingProperties()
Injects
LogServiceModule
+ database: MongoDB
+ reader: COBIDReader
+ client: MongoClient
+ provideMongoClient(Credentials): MongoClient
+ provideMongoDB(MongoClient, String): MongoDB
+ provideCOBIDReader(String): COBIDReader
Handles
<Interface>
LogService
+ putRecord(DataRecord): Type
+ putRecords(List<DataRecord>)
Parse files
DataParser
+ client: LogService.Client
+ sendData(File)
1...*
LogServiceHandler
+ database: MongoDatabase
+ reader: COBIDReader
+ truckCache: Map<String, List>
+cobidCache: Map<String, String>
+ putRecord(DataRecord)
+ putRecords(List<DataRecord>)
+newTruck(Document)
+newCOBIDforTruck(Document)
ToDatabase
DataRecord
+ Date: long
+ Can_ID: int
+ COBID: int
+ truckID: String
+ Bytes: List<Byte>
+ getters(Type): Type
+ setters(Type, Type): Type
Parses Bytes
Converts Data
COBIDReader
+readData(DataRecord) Document
1

Fig. 13: Data Logger Conversion Service Domain Model



```
{
  "206" : [
    {
      "desc" : "CommandAll",
      "dataLength" : 2,
      "type" : "unsignedInt",
      "divisionAdjust" : 1
    },
    {
      "desc" : "CommandSpeed",
      "dataLength" : 2,
      "type" : "signedInt",
      "divisionAdjust" : 1
    },
    {
      "desc" : "CommandAccelerationChange",
      "dataLength" : 1,
      "type" : "ubyte",
      "divisionAdjust" : 1
    },
    {
      "desc" : "CommandDecelerationChange",
      "dataLength" : 1,
      "type" : "ubyte",
      "divisionAdjust" : 1
    },
    {
      "desc" : "VoltageSetLevel",
      "dataLength" : 2,
      "type" : "decimal",
      "divisionAdjust" : 100
    }
  ],
  "186" : [
```

Fig. 14: COBID configuration for COBID 206.

6 References

1. Iowa Department of Transportation. “Track a Plow” http://iowadot.maps.arcgis.com/apps/webappviewer/index.html?id=3d5bc4ec8c474870a19c7e8f44b39c9c

2. “US Xpress Inc - Get Your Freight Shipped - U Can Depend On U.S.” *USX Corporate*, www.usxpress.com/.

3. Van Rijmenam, Mark. “Trucking Company US Xpress Drives Efficiency with Big Data .” *Datafloq*, datafloq.com/read/trucking-company-xpress-drives-efficiency-big-data/513.