Data Analytics for Snow Plow Truck Data

Design Document

Team 23
Client: Henderson Products
Adviser: Dr. Goce Trajcevski
Alex Smola/Design Lead, Alan Peine/Git Master, Colin Heinrichs/Test Engineer, Evan Warych/Report Manager,
Michael Entin/Meeting Reporter, Zach Wilson/Meeting Facilitator & Communication Lead
sdmay18-23@iastate.edu
http://sdmay18-23.sd.ece.iastate.edu

Revised: 12/03/17 Version 3

Table of Contents

List of Figures

List of Tables

1 Introduction

The main goal of the project is to help with the visualization of data that is being received from sensors on a group of snowplows. We also discuss the environment this project will operate in, who and what this project is for and our expected end product. We also lay out a set of limitations and assumptions we have laid out for our project.

1.1 Problem and Project Statement

Currently the data is being sent from these trucks containing information about the plow including its coordinates, fuel consumption, and select statistics about the plows performance. That data is then being stored on a server, but is currently in an undesired format to be analyzed. It exists as 1 data entry per line that needs to be inputted into Excel and manipulated for around an hour before it is human readable. This still has no analytics and has to be done for each file. The Data Logger also doesn't perform any kind of data redundancy check or compression. This results in large files being sent that contain only zeros.

Ultimately the product will be a web application that will display these statistics about the truck that can be studied to forecast the trucks performance in the future. With this knowledge, quality issues can be spotted before they affect performance of the truck, saving money from a potential breakdown.

The solution to this as previously stated is to create an application that hosts and helps visualize the data being received. At its current state the data is being stored on a server hosted by Henderson Products. So our application will be constantly pulling new data as it makes its way to that server, and will convert it into a readable format to be stored in our database. We will then design an interface that will allow the info to be easily read and analyzed. Most likely the product will allow clients to log on and allow them to view and monitor their own trucks and data.

1.2 Operational Environment

As our project deals with transferring, converting, storing, and accessing data; we don't expect our product to be exposed to any notable conditions. We expect our end product to run on one or more servers, and be 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. It's important to note that while the data will be coming from loggers on Henderson vehicles, the actual capturing of data is already implemented and is considered out of scope for this project.

1.3 Intended Users and Uses

The intended users of the CANBus data app will be the good employees of Henderson products and possibly even their clients. The clients using the web app, however, is a bit of an assumption (as stated in the next section). 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 half 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, 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 able to withstand the tests of time and the harsh environment of winter.

1.4 Assumptions and Limitations

In order to define the scope for our project we set limitations for our project. This allows us to clearly lay out what is expected of our team from ourselves and our client. We've also defined a set of assumptions that are subject to change should a need arise.

1.4.1 Limitations

1. The project will be completed in its entirety by May 2018.
2. The project will not require any hardware design.
3. The project will not use more cellular data than the initial transfer from vehicle to server.
4. No data will be lost in transformation or translation.
5. Data will be moved to the web application within 24 hours of creation.
6. Adding a new user will take less than 2 minutes.
7. Clients will only be able to view data from their own vehicles.

1.4.2 Assumptions

1. We will receive all information on how data converts from hexadecimal bits into relevant data.
2. We will receive access to Henderson Products server in order for us to be able to pull files from it and convert.
3. We will be able to contact our client for important information within a reasonable amount of time.
4. The web application will have users with unique logins to guarantee security and data integrity.
5. Vehicles will each have their own unique identifier.
6. Clients of Henderson Products will be able to view data sent from their own vehicles
7. The maximum amount of simultaneous users shall not exceed 100.
8. The web application will be a single page that allows the manipulation of data to be viewed and read effectively and efficiently.

9. The web application will adhere to the branding of Henderson Products
10. The completed product will not be viewed outside of the United States
11. The website will only need to be in English.
12. New desired features will be given with the understanding that time may be a factor on successful implementation.
13. We will not need to develop for the sensor, all data being sent is all data needed.

1.5 Expected End Product and Deliverables

Our end product can be separated into 3 main parts. We will have a Data Publishing Service, Database, and Web Application. An expected delivery date for each of these pieces is states in Section 1.5 or our Project Plan.

Data Log Parser

The data log parser will take the log files from the trucks, parse them, and return the data in a more manageable form. Initially this parser could return a human readable file. However, the end goal for the parser is to take the data it returns and insert it into a database.

Database

We expect to deliver the design for the database that will allow our client to store, access, and organize the truck data. The database will be populated with data extracted from log files by the parser. The database will be accessed by the client through the web application.

Web Application

The web application will be used by the client to view the data being sent from their trucks. The application will offer different ways for the client to view the information such as: graphs, maps, and tables.

2. Specifications and Analysis

In Section 1.5 we broke our end product into 3 main parts. In this section we split it into 5 main design aspects. Each of aspects has multiple solutions and will be analyzed for the best solution below.

2.1 Proposed Design

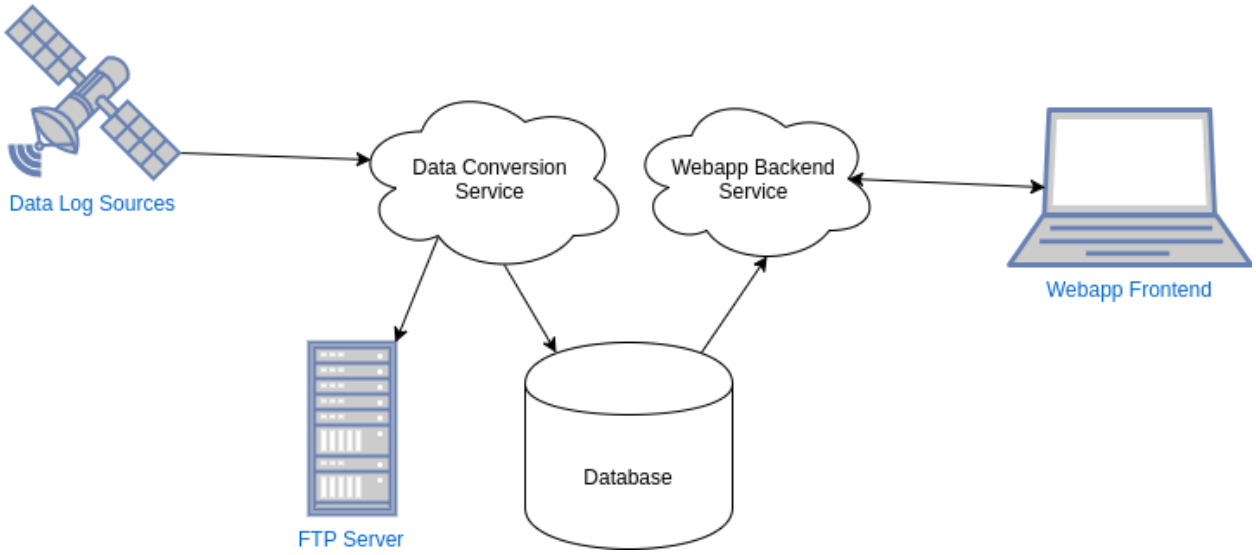

Fig. 1: Design diagram describing the layout of the different parts of our application

Our team has decided to implement a web app to solve the problem of having to convert the CANBus data by hand. You can refer to the Figure 1 as I describe the different parts of our design. In order to handle data conversion and storage, we plan on creating a web service that will convert and store the data logs in our database; which will be called directly from the data loggers. This meets the data conversion functional requirement. From there, we will store the data in a mongodb database in order to meet the data storage requirement. Our webapp will consist of a backend service and a javascript frontend. The backend service will be implemented as a RESTful service using Spring boot, which will provide the frontend with and api for querying truck data from the database, meeting the responsiveness non-functional requirement. Our frontend will be written using the ember.js framework in order to meet the data analysis requirement. By doing this we can make use of frontend graphing libraries that will allow users to better visualize the data. In order to meet our security requirements, we will implement an authentication service for logging in and accessing the data. All justifications for these choices can be found in the design analysis portion, and a more complete block diagram can be found in the appendices section (Figure 9).

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.
Performance 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. We also want to ensure that data is available within 24 hours of being logged.
Throughput/Scalability Requirements - Our system will need to be able to handle variable amounts of data during different parts of the year. It should be able to scale easily to handle more data.

2.2 Design Considerations:

As said in Section 1.5 our project can be broken down into 3 parts. We considered a number of technological solutions to each of these parts and analyze them below.

- Data Log Parser:
    - ANTLR
        - A popular language parser generator
        - Might be too advanced for what we need
    - Build our Own Parser
        - Can use a language that we know well
        - Does not need to be complex
        - Can set up configuration files to allow different kinds of data to be recognized

- Data Storage:
    - SQL database
        - Familiar with the syntax and technologies
        - Short setup time
        - Consistent & reliable
    - Microsoft SQL Server

        - Many powerful data analytic capabilities
        - SQL queries make for easy use
        - Henderson already has access to this database
    - Data Warehouse
        - Works better with companies in general
        - Can store large amounts of data
        - A new concept most of us aren't familiar with
    - CouchDB
        - Works well with web applications
        - Not great with handling large amounts of data
    - MongoDB
        - Non-relational, which may allow for horizontal scaling
        - Can work much faster than an SQL database
    - Apache Cassandra
        - Designed to work well with large amounts of data
        - Also Non-relational
        - No experience from team working with it

- <u>Web Application:</u>
    - Frontend
        - AngularJS
            - Uses a MVC architecture for developing frontend applications
            - Experience from multiple team members
        - ReactJS
            - Uses uni-directional data flow for organizing components
            - Organizes views into reusable components.
            - Makes it easy to port to mobile with React Native
        - EmberJS
            - Front end javascript framework
            - Handles routing, templating, and data-fetching
            - CLI makes it easy to initialize and configure applications
    - Backend
        - NodeJS
            - Can have full stack javascript
            - Some of our team has exposure to Node
        - Laravel Framework
            - PHP is usually not fun to code in
            - The framework is really great & can make some good looking apps
            - May not have the graphical functionalities we're looking for; can't be used to create single page apps on its own.
        - Java Spring
            - Nearly all of our team knows Java
            - Used widely, has great documentation

- Apache Thrift
    - RPC framework that won't limit us in choice of language.
    - Gives us more freedom in terms of software design.
    - Forces us to define our service interfaces and the models that they work with, leading to a better design.
    - Abstracts away communication between software modules. (In our case the client and server)

2.3 Design Analysis

Separating the solution into three distinct parts allows for us to develop and test the different parts concurrently. It also allows us to be able to use different technologies for the different components (such as a php backend and a angular frontend). We then have to option to switch out technologies without having to rework the other components.

The proposed design was created after identifying 5 different design aspects needed in order to meet the requirements defined above. These aspects are: Data Loggers, Data Ingestion, Data Storage, Webapp Backend, and Webapp Frontend. How the fit together can be seen in Figure 2.

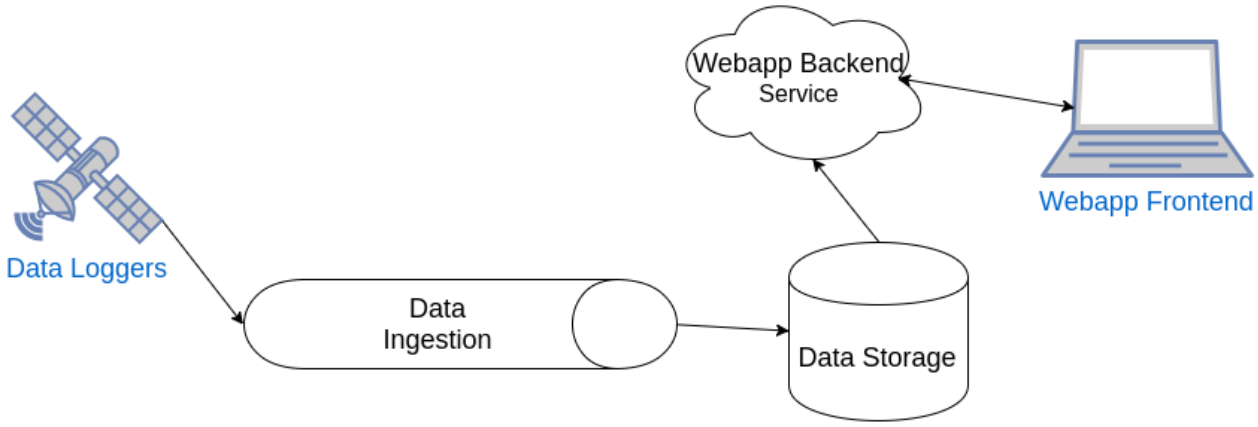

Fig. 2: Our Proposed design diagram displaying the needed services at each part.

These design aspects form the different parts of our design, and each have required some decisions to be made to meet our requirements stated above. These decisions are outlined below:

2.3.1 Data Loggers

Currently data is being hosted on an ftp site in the form of log files filled with hexadecimal values. This data needs to be converted to make the values easier for our client to process and understand. We are also looking to eliminate data received that returns values of zero.

We plan on rewriting some of the code running on the data loggers to facilitate our plans for data ingestion, which we go into more detail in the next subsection. We also plan on changing the code some to control what data is sent so that we can prevent redundant values from being published.

2.3.2 Data Ingestion

We have 4 ideas on how we plan to handle data ingestion. Data is logged whenever a truck is in service. Once a file reaches a certain size the data logger sends the file to an ftp server once a reliable connection can be established.. The current formatting of the data is a single packet per line. Each packet is labeled by a by either a Communication Object Identifier(COBID) [2] or a Parameter Group Number(PGN) identifier [1]. What the label depends on what communication protocol was used to transmit the data. PGNs are part of the j1939 standards and are used when the data logger is transmitting GPS data and GPS related controls. COBIDs are used for all other forms of data. The data packets remain the same size and format. This format was originally designed for the transmission of COBIDs. COBIDs are a 3 digit decimal identifier. PGN identifiers are an 8 digit hexadecimal value. Below are 3 possible designs for the implementation of data ingestion.

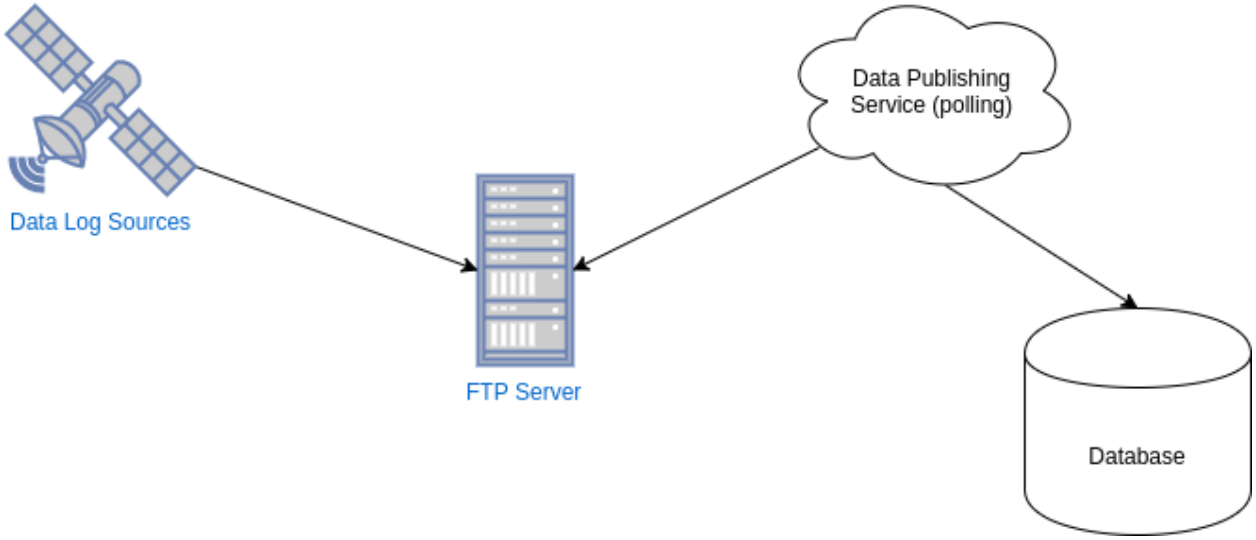

Fig. 3: Proposed design diagram #1 of the connections between the Data Log and Database

Design 1(Figure 3): In this design we are leaving the data logger mostly as is. It is still being sent in 1 packet per line format and stored onto the FTP Server. We change the naming scheme for files on the FTP Server to each Vehicle's Identification Number(VIN). This allows us to have a more universal and unique format than the current scheme of just using a human defined vehicle name. We would also change the data storage from all readings to only the readings that are different from the last. This will help us limit the amount of data we are sending. We would then create a Data Publishing Service(DPS) that polls the FTP Server for new files. Each file would have its hexadecimal data be converted to a more relevant form ie. byte, integer, float. We will then wrap the data into a json key-value pair so that database can read it when the service sends the data.

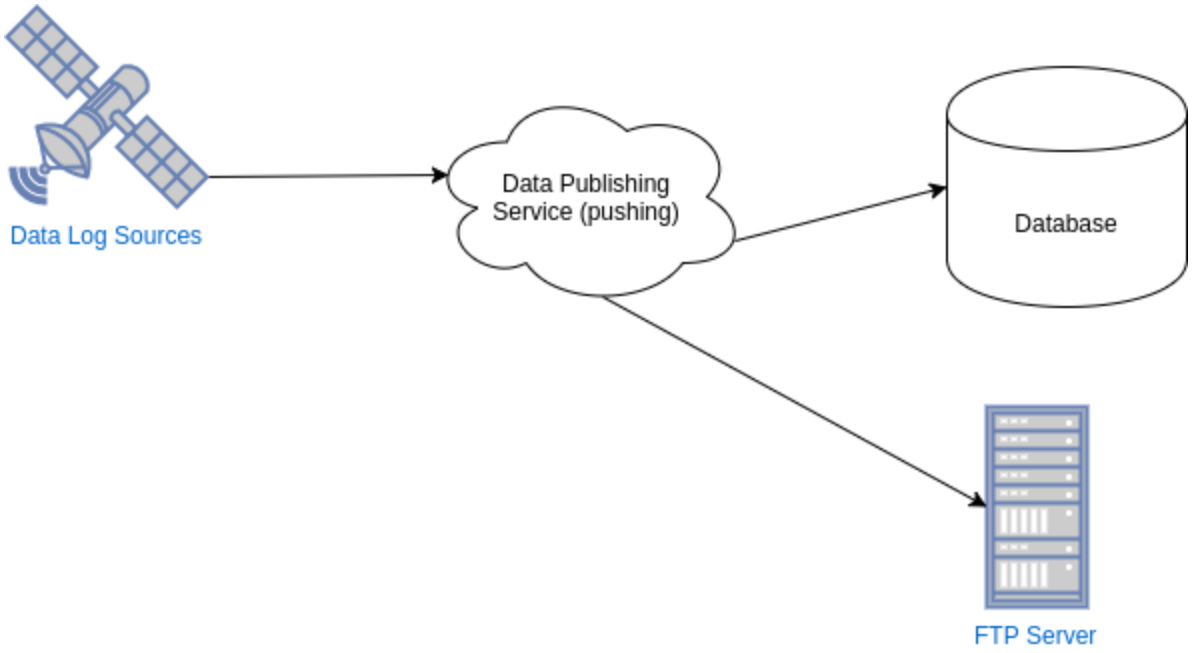

Fig. 4: Proposed design diagram #2 of the connections between the Data Log and Database

Design 2(Figure 4): We would first add the data compression to the data logger. Then when data is ready to be sent, the data logger would call the DPS. The DPS would forward the Data to the FTP Server to be stored as is. We would then prepare the data the same way we did in Fig. 3. This design allows us to have data be sent directly to the database when it's ready rather than polling it from the FTP server. We prototyped a service that would work in this way.

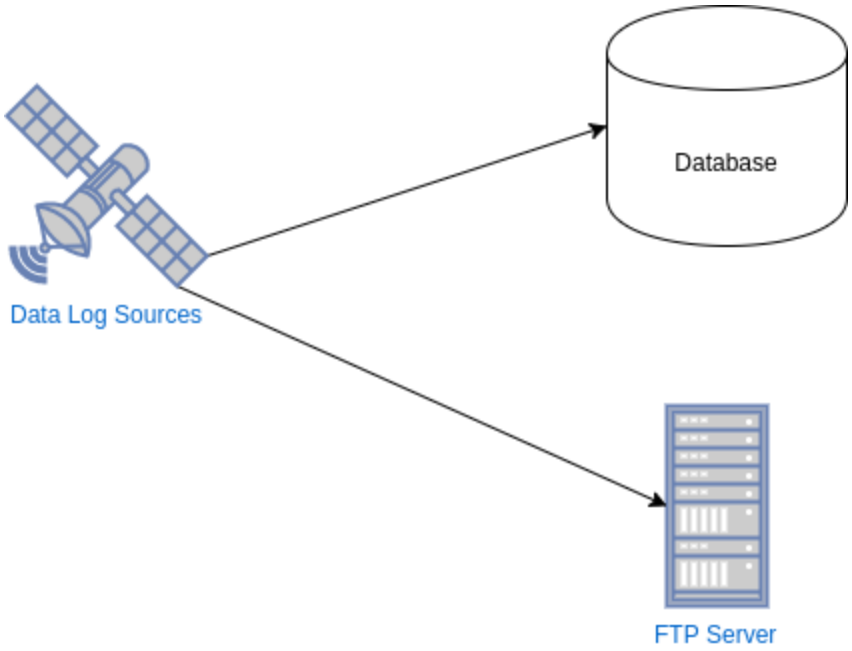

Fig. 5: Proposed design diagram #3 of the connections between the Data Log and Database

Design 3(Figure 5): This design involves the most change to the data logger. We would still add in the data compression. We would also leave the current way the data is wrapped and send it to the FTP Server. We could then add conversion to json key-value pairs on the data logger. This data would be send directly to our database.

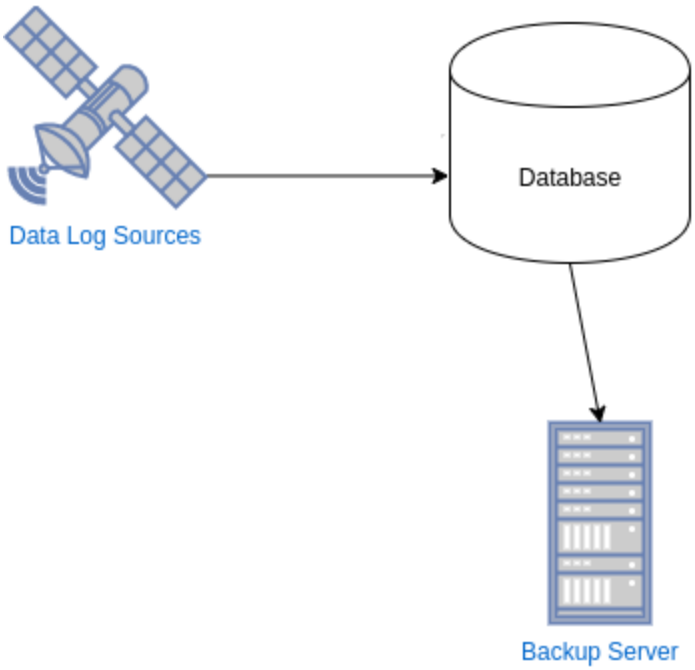

Fig. 6: Proposed design diagram #4 of the connections between the Data Log and Database

Design 4(Figure 6): This design seeks to limit the use of the FTP server and tread it more of a backup of the database. The Datalogger would be changed the same as as in Figure 5. However, we would remove the conversion into the FTP Server format. The data is sent directly to the database in json key-value pairs. Once a week we would backup up the database to what used to be the FTP Server. This way would eliminate the need for DPS. The data on the Backup Server would either be a snapshot of the database or storage of all the data on the database in the same way it was added in.

Our ideal design is displayed in Figure 6. This design removes the need for DPS and limits the amount of data transmitted. However, our client wishes to keep the data stored on the FTP server as is and we don't want to send the data more than once using a mobile data connection. This limits our potential designs to either Figures 3 or 4. We feel it will be simpler to have the data logger call DPS and send the data that way then to poll the FTP Server to check for new files. Thus we have decided to implement the design displayed in Figure 4.

2.3.3 Data Storage

Data is the bulk of our project. How we plan to store it and access it is a large part of our design. In this subsection we analyze different types of databases and the data scheme we have designed.

2.3.3.1 Choosing A Database

SQL vs. NoSQL

Henderson will be working with an ever growing amount of truck data. They log a substantial amount of data every day for just a single truck, so adding even more on top of that will require a lot of space. A reasons why a NoSQL option would be more beneficial is because of horizontal scalability, or the ability to create more space for data simply by adding additional servers or cloud instances. Additionally, we can opt to create key-value storage options that function very similarly to SQL databases without the vertical scaling drawbacks. Secondly, NoSQL databases have been proven to work well with AWS hosting, which is something Henderson has asked us to look into in case they can't get us any server space due to security reasons. Many successful businesses such as Netflix, Reddit, and many others use a combination of NoSQL databases [3].

SQL Options

The SQL options we can choose from are regular SQL, PostgreSQL, and a Sequel Server. The benefits of a sequel server over regular SQL include advanced python analytics, machine learning, adaptive query processing, along with many other benefits which may not be very useful for this application in particular. Along with those benefits, Henderson already has access to a Sequel Server (which would normally cost thousands of dollars to purchase). This makes using the Sequel Server an obvious choice over regular SQL, as everything that SQL can do, the Sequel Server can as well. However, we still have not looked at PostgreSQL. As you can see on the charts below, the spreader and conveyor data have some of the exact same data fields (speed, motor temp, data temp, etc...). An advantage PostgreSQL has over the other two options is that it has inheritance capabilities [4]. This would be of use to us to reduce redundant data and keep things clear.

NoSQL Options

The NoSQL options we've decided to choose from are Apache Cassandra, CouchDB, and MongoDB as they are some of the most commonly used NoSQL databases we could find. CouchDB has the advantage of being designed for use with web applications in mind. It, "... enables applications to store data locally while offline, then synchronize it with CouchDB and compatible servers when the application is back online, keeping the user's data in sync no matter where they next login." [5]. It is also compatible with many browsers, is lightweight, and open source. This would be nice to use considering the one interface Henderson plans to use this on is on a browser. Apache Cassandra's biggest benefit is that it handles large amounts of data seamlessly. This would be an important thing for ust to consider based off of the tremendous amounts of data Henderson's trucks will be producing during the snowy season. MongoDB seems to be a nice mix of CouchDB and Apache Cassandra. It can communicate via JSON (which is preferable for websites) while simultaneously being able to handle large volumes of data [6]. Along

with being easy to use and an additional benefit of not requiring a database administrator, MongoDB seems to be a very strong option for our application at this point.

Final Decision

Considering the two macro options (SQL vs. NoSQL), our team would prefer to use NoSQL due to the immense benefit horizontal scaling would provide to Henderson Products. It seems that we would get a higher variety of benefits from MongoDB and that would be our first choice for a NoSQL database. It may be beneficial for us to consider using PostgreSQL in order to incorporate inheritance into our database structure to reduce redundancies, but the benefits of an already paid Sequel Server would be the best option. If Henderson chooses not to go with a hosting service, we may have to choose that as our database option.

2.3.3.2 Data Scheme

We have a good idea of what data is being sent from the data loggers and how they are related to the different mechanical aspects of the snowplows. We have put tables in the appendices (tables 1-4) describing the different schemas that will be needed in our database relating to that data. We will also need credential data for any users using our database to allow for logging in and a list of all of Henderson's clients in order to display who owns which trucks.

2.3.4 Webapp Backend Service

Architecture Style

The first design decision that needed to be made for the backend services was the general architectural style of the web service. When researching the potential options, the debate seemed to be narrowed down to following HTTP with XML/SOAP or another more flexible protocol like building a REST structure. SOAP initially seemed to be more beneficial for our application, as data manipulation from the client will not be extremely prevalent in the product, but rather simple requests to provide end users with the data, a rigid protocol like SOAP would more easily ensure the proper transportation of this data. However, REST eventually became the choice for the backend service for a couple of reasons. The first of which pertains to the amount of data that will be transferred. With so many options available for the end client to analyze their trucks data, the lightweight form of a REST structure with JSON objects allows for a more responsive application. Secondly, with REST on the rise in web applications the amount of support and frameworks available provided developers with extra tools to ensure a proper web service.

Language

The second challenge for design of the backend service was settling in on a language to develop in. With many members of the team having background developing web applications in either of Java and Javascript, these were the two languages that were focused on in researching which option would be best for the application. It quickly became evident that Java provided more benefits to our needs than Javascript did. Again one of the main qualities we desired was support for the chosen language. With Node.js being just 8 years old, it is relatively young compared to most Java utilities, thus having less available tools to aid in our development. Secondly, as Henderson Products will be providing this application to other businesses, security and reliability was another

major concern for us in the decision. Many utilities were available for Java applications to remedy this concern, such as Spring's security framework which can help us keep sensitive data safe.

Framework

As previously stated, the last design decision that was kept in mind during research was the framework to build the service off of. Constraints for the framework were that the framework should be well supported, provide structure for a REST api, provide secure data transfer, and ability to transfer large amounts of data. The frameworks that made it to final considerations were Spring and JAX-RS. Both frameworks provided plenty of well documented support to create a REST api, with the latter even being included in Oracle's Java EE 6. The next major factor was security of the data. JAX-RS offered authorization and authentication, the same as Spring's Security feature. However, Spring Security went further with features that seemed to be valuable in an enterprise environment to not make it as tedious for the end client. Furthermore, Spring Boot provides an easy format for packaging all the extra utilities Spring has to offer for our application. For instance it could package the Spring MVC, Spring Security, and Spring Data for easy deploys to the server. In the end Spring just offered more for our application to be built upon, and was the choice for the backend framework of the product.

2.3.5 Webapp Frontend

There are a number of javascript libraries and frameworks that can be used to build single page web applications. React, Angular, and Ember are three of these frameworks/libraries that we considered for this project.

React

React is the simplest of the three options. It is primarily a library built to handle the views of an application. This means that it must be paired with additional libraries if we want to have features such as routing.

Angular

Angular, unlike React, is a full-featured framework. This means that it handles everything from manipulating the DOM to AJAX calls. It is built for CRUD applications (applications with the primary purpose to create, read, update, and delete information) and does not fit well to applications of other models.

Ember

Ember, just like Angular, is a full-featured framework for developing websites with rich user interfaces. It provides a robust development toolkit to help develop applications, routing capabilities, a templating engine, and a data layer that provides a consistent way to communicate with external APIs.

Final Decision

Our final decision came down between Ember and Angular. Both provide features, such as routing and data manipulation, that React does not. In order to get those features with React, we would

have to use additional libraries which makes learning more difficult and initializing the project even more tedious.

Both Angular and Ember are incredibly similar. Both provide a Command Line Interface tool to help developers initialize, serve, and test applications. This is incredibly useful as setting up javascript projects can be incredibly tedious at times. Both frameworks offer great documentation and guides on their websites and since both of them are open source, there are numerous libraries that are compatible with each framework. One of those being Chart.js a graphing library our team was looking at to use to display the truck data in various graphing formats.

Either Angular or Ember would be suitable for the needs of this application. However, Ember edges Angular out for a couple of reasons. Ember 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 like Ember's best practices and syntax more than Angular. We believe that Ember will make developing our single page application easier.

3 Testing and Implementation

3.1 Interface Specifications

Given the fact that the dataloggers are already configured, working, and tested; we don't plan on making any hardware modifications for our project. As such we have no need for hardware testing, and our testing will exclusively involve software tests, which we will split primarily into functional and nonfunctional tests.

Unit tests will be created to test small units of functionality in our software, and will ran both during development and before being pushed into the main branch of our repository.

Integration tests will be designed to test the connections between different modules of our system -- e.g. Database, backend service, frontend, etc. -- to ensure that they work as expected. In order to facilitate our integration tests we will need to build a separate copy of our production system that will act as a pre-production stage. This will provide us with an environment to test changes and ensure correctness before they make it to production.

Functional Testing

Data Conversion Testing - To ensure that our data is being converted correctly we plan on writing unit tests that utilize a number of different unit tests.
Data Storage Testing - We plan on creating integration tests to ensure that our connection to the database doesn't break as we make changes. We will also create some unit tests that mock the database connection and ensure it is being called correctly.
Data Access Testing - It will be important for us to test our webapp backend to ensure that all the endpoints are functional. This will involve some unit tests for some of the operations being performed and integration / manual testing to ensure that the service endpoints can be reached.
Data Analysis Testing - Our frontend will involve a lot of manual testing during development to ensure everything is being displayed correctly. We also create unit tests to ensure operations are being performed correctly on the data.

Nonfunctional Testing

Security Testing - To ensure access to our system is restricted to Henderson and their clients, we plan on performing manual testing of our endpoints to ensure that they are locked down to users with correct credentials.
Responsive Requirements Testing - We plan on setting up logging / metrics for our backend service to help us determine bottlenecks and areas where we need to improve to meet our performance requirements.

3.2 Hardware and Software

To perform our functional and Nonfunctional tests we plan on using the following Software Tools:

Junit

Junit is a testing framework for Java development that is widely adopted both in industry and in academics. It provides easy to use test fixtures that make it easy for developers to write and run unit tests. We plan on using Junit (or a similar testing framework if we decide to use a language besides Java) while developing our backend services.

Mockito

Mockito is a framework for creating mocks and stubs for Java objects and methods respectively. A mock of an object is a "dummy" object that is used while writing unit tests to isolate units of software. Mockito makes it easy to create mocks of objects and to stub out their methods to return something predetermined by the tester. This becomes useful when the unit of software under test makes a method call that we aren't testing. By mocking the object that is being called and stubbing the method call we can ensure that our tests are consistent when they are ran.

QUnit

QUnit is a framework for writing unit tests for javascript. This will help us ensure that our web app is working well on the client side.

3.3 Testing Process

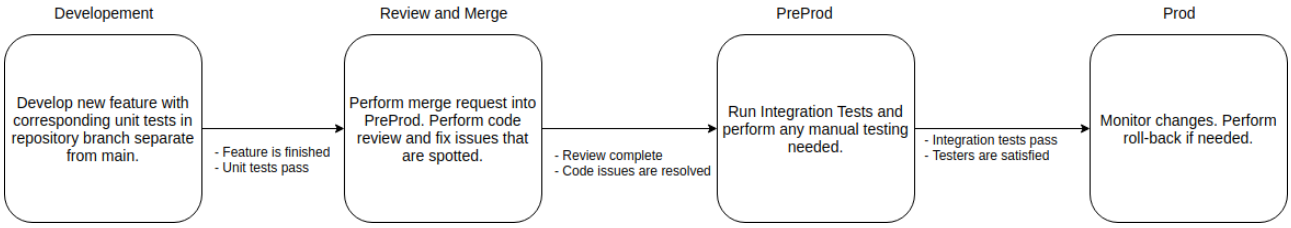

Fig. 7: Proposed Development Testing Process

Figure 7 outlines our process for testing and ensuring code quality. It is split up into four stages: Development, Review and Merge, Pre-prod, and Prod. The process begins in the "Development" stage with a new feature that is being developed. In this stage the developer creating the feature will write the code implementing it, as well as an appropriate number of unit tests -- which need to pass before the "Review and Merge" stage can be reached.

In the next stage the code that has been developed is looked over by another team member. This is achieved by performing a merge request into the pre-prod branch of our repository. Once the merge request has been made a code review needs to performed to verify the code change is satisfactory. Once another team member has accepted the merge request the change can then be

merged into the preprod branch of our repository; leading us into the “PreProd” stage of our testing process.

The Preprod stage is where we will test our changes in an environment that is similar to our production stage. It exists to allow us to spot bugs before they can have negative effects such as downtime or loss of data. It is also where we will perform integration tests between the different modules of our system. Once the integration tests pass and the testers are satisfied, the change can then be pushed into production.

3.4 Results

So far the main results of our project are prototypes, issues that have arisen while designing, and ongoing challenges we have encountered.

3.4.1 Model and Simulation

Prototypes

As we’ve been mostly focusing on design and planning this semester, we have only gotten around to implementing a couple prototypes.

Data Conversion Service

This prototype demonstrates the data conversion service we plan to use to convert the data received from the data logger into meaningful data which can be stored in a database. The following is a link to a video demonstration of our prototype at work:
https://iastate.box.com/s/x8hvbutv5x137rdf5qcpq8dnpjxq9n2h

Wireframe

In figure 8 is a screenshot of a wireframe of the main section of our application. It will allow Henderson products to view the gps data of trucks and all of the data regarding spreaders, belts, and other truck related data. Users would be able to login and view all of the data they have access to.

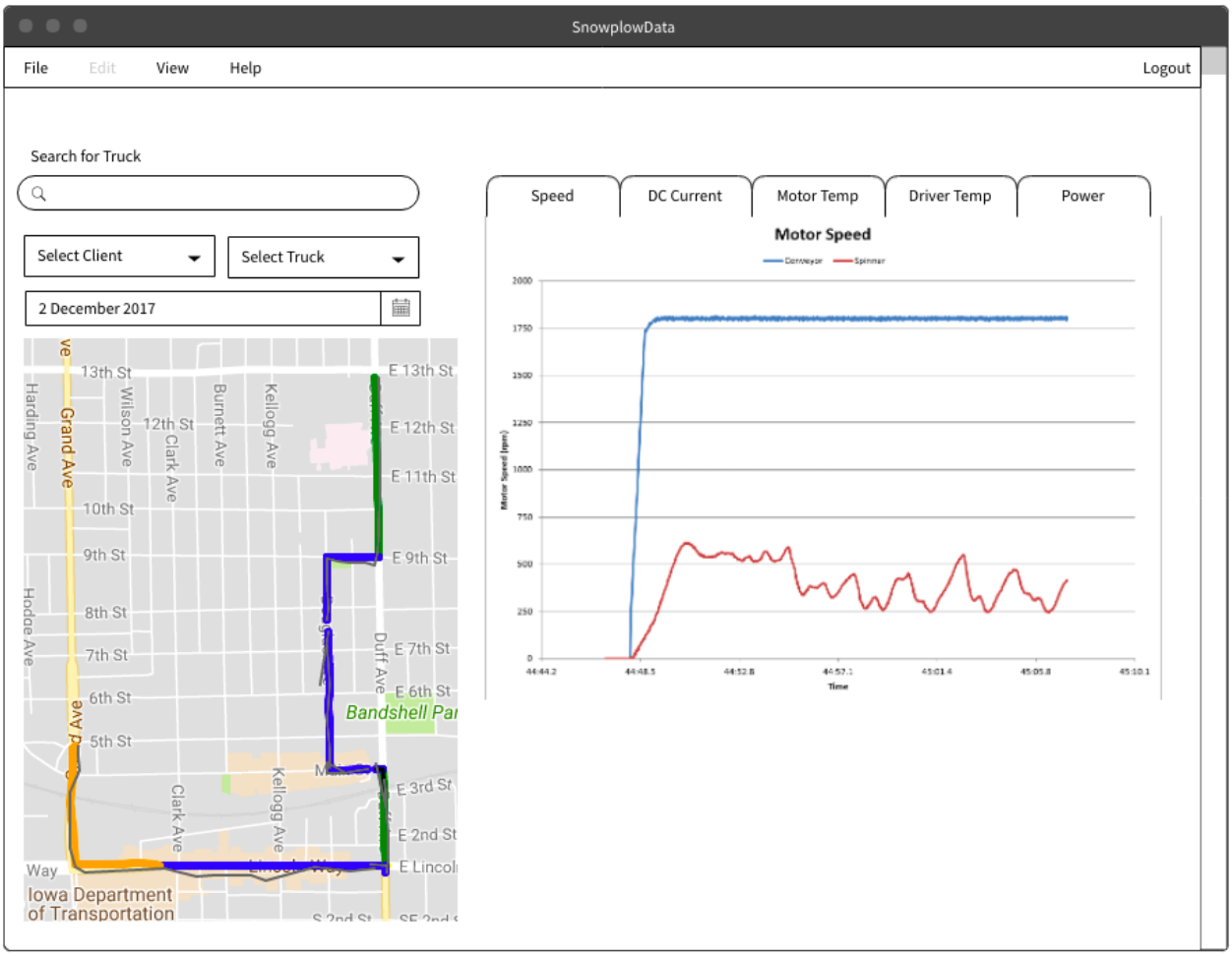

Fig. 8: A wireframe of the home page of the application. It contains search functionality and date selection for GPS data.

3.4.2 Issues

We have been dealing with two main issues throughout this semester. Server Space and GPS data are ongoing roadblocks for development, but will be resolved shortly.

Server Space

One of the issues we've encountered during the implementation of our project has been working with Henderson on determining exactly how we will host our application. They have been going back and forth on whether they can allow us to use their server space or if we should be looking towards using a hosting service such as AWS or Microsoft Azure. We are fully prepared to start working on the project once they have decided which is most secure for Henderson, but it is tough to get to work without somewhere to host our database.

GPS Data

Another issue we've had to deal with is the possibility of incorporating GPS data with some of the data loggers Henderson uses. They wish to have that data available to them, but had taken a long

time to set up data loggers with that data since that option had been disabled in the loggers by a previous employee of Henderson. We finally have our hands on the GPS data format and have started working on thinking of ways to store that data.

3.4.3 Challenges

We face a number of challenges in our project. Our first one is how we should handle GPS data. We've done some research towards what the best solution toward our GPS data points will be. We read a few things on low and high sampling GPS data. We're hoping that our data will be frequent enough for Google Map's API to interpret and place on roads reliably. Next is how we're going to handle queries to our database. We have a large amount of incoming data. We need to figure out an effective way to store this data so that queries are easy. We need to be able to display the data in any way possible in order our client to see how everything relates. We want to avoid writing complex code in order to handle this. We've researched Data Warehouses as a potential solution to this challenge. Lastly has to do with some of our groups background. We have a software oriented project. The computer engineer and the electrical engineer in our group each have their own respective learning curves.

4 Closing Material

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

4.1 Conclusion

The goal for our project is to make it easier for our client to understand the data that they are receiving from their trucks. Currently, the information is being stored on a server hosted by Henderson Products, and then manually converted from hexadecimal values. We are going to be creating a web application that will help convert this information faster. This information is being used to predict the performance of each individual truck and will help the clients to better monitor their own trucks. We will be formatting the information that they are receiving so that they will be able to see the quantities of each piece of information over time. Our design consists of 3 main parts that can be further broken down into 5 design aspects. These are the data loggers, data ingestion, database, webapp backend, and webapp frontend. During the next semester we will work to implement all 3 parts of using the design decisions we made in these 5 aspects.

4.2 Acknowledgement

Thank you to James Timmermann 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.

5 Appendices

Currently we already have 1 appendix, but we are open to adding more should the need arise.

5.1 Data Schemas

The following tables depict the expected names and types of all the data received from the raspberry pis that will need to be stored in our database.

COBID 0x186			
Status Word : U8	Actual Speed : S16	Motor Temp : S16	Drive Temp : S16
COBID 0x286			
Torque Current : S16	Filtered DC Current : S16	Filtered DC Bus Voltage : U16	DiginAll : U16
COBID 0x386			
DC Motor State : U8	Output Voltage : S16	DC Motor Current : S16	

Table 1: Spreader Data Schema depicting what data object will need to be in the database

COBID 0x187			
Status Word : U16	Actual Speed : S16	Motor Temp : S16	Drive Temp : S16
COBID 0x287			
Torque Current : S16	Filtered DC Current : S16	Filtered DC Bus Voltage : U16	DiginAll : U16
COBID 387			
Vehicle Speed : U16			

Table 2: Conveyer Data Schema depicting what data object will need to be in the database

COBID 09F80100	**Position - Rapid Update**			
Latitude : 32 bit FLOAT	Longitude : 32 bit FLOAT			

Table 3: GPS Data Schema depicting what data object will need to be in the database

COBID 351				
Pressure Transducer - Left Tensioner Cylinder (PSI) : U16?	Pressure Transducer - Right Tensioner Cylinder (PSI) : U16?	Pressure Transducer - Conveyor Motor (PSI) : U16		
COBID 352				
Tailgate Lift : bool (index 1)	Tailgate Lower : Bool (index 3)	Conveyor Unload : Bool (index 5)	Conveyor Reverse : Bool (index 7)	
COBID 353				
Conveyor Speed (RPM) : U16				
COBID 354				
Proximity Material Sensor : bool (index 1)				

Table 4: Blackbelt Maxx Datalogger Data Schema depicting what data object will need to be in the database

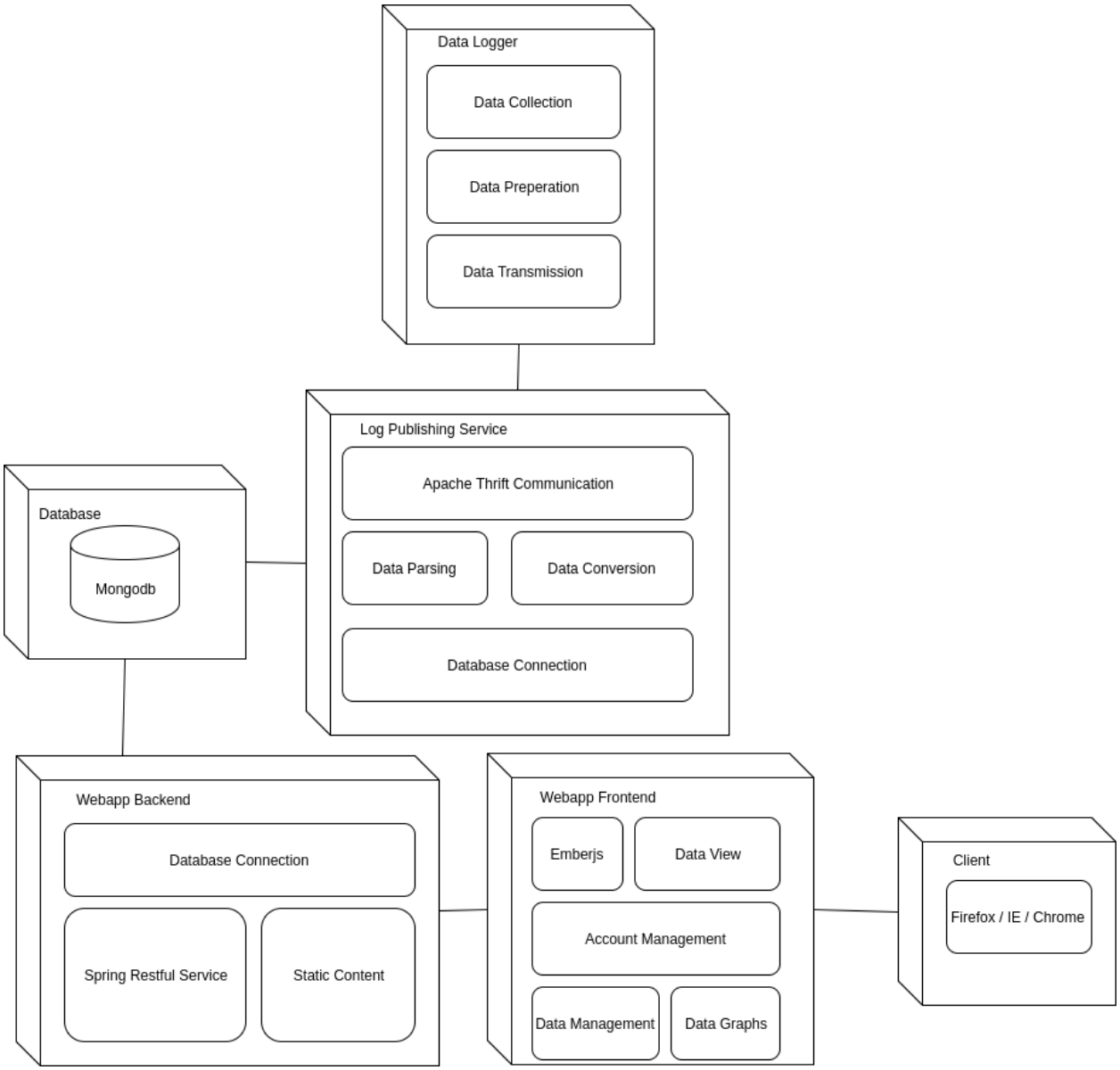

Fig. 8: Proposed System Block Diagram

6 References

[1] Society of Automotive Engineers, "The SAE J1939 Communication Systems," 26 11 2017. [Online]. Available: http://www.sae.org/misc/pdfs/J1939.pdf.

[2] CAN in Automation, "CANopen – The standardized embedded network," 20. [Online]. Available: https://www.can-cia.org/canopen/. [Accessed 26 11 2017].

[3] "NoSQL Databases Explained," MongoDB, Inc., 2017. [Online]. Available: https://www.mongodb.com/nosql-explained. [Accessed 28 November 2017].

[4] "PostgreSQL," Wikimedia Foundation, 26 November 2017. [Online]. Available: https://en.wikipedia.org/wiki/PostgreSQL. [Accessed 28 November 2017].

[5] The Apache Software Foundation, "CouchDB," The Apache Software Foundation, 2017. [Online]. Available: http://couchdb.apache.org/. [Accessed 28 November 2017].

[6] Redis Labs, "Top Five NoSQL Databases and When to Use Them," Quinstreet Enterprise, 2017. [Online]. Available: https://www.itbusinessedge.com/slideshows/top-five-nosql-databases-and-when-to-use-them-04.html. [Accessed 28 November 2017].