CS Capstone Project Hand Off - All Content

June 6, 2019

HyperRail App

Prepared for

OPENS Lab

Chet Udell

Prepared by

Group 25

Vincent Nguyen
Adam Ruark

Abstract

This document is a compilation of all written work put into developing the HyperRail Application.

CONTENTS

1 Introduction

The HyperRail application is a web-based application used to manage and control a rail-based sensor package. This project was requested by Chet Udell at the OPEnS lab, located at Oregon State University. It was requested because the rail-based sensor package (also known as the HyperRail) required an updated user interface to control it. The legacy UI had several limitations to it, so this project was created to start from scratch and implement a more modern UI with the ability to improve it in the future. Although our primary client for this project was Chet Udell, we mainly worked with a couple of employees at the OPEnS lab: Manuel Lopez and Lars Larson. Manuel was the creator of the HyperRail and introduced us to how the system works, while Lars is a primary user of the device and had a large say in the required features for this project.

For this project, our team comprised of three members originally (Adam Ruark, Vincent Nguyen, and Yihong Liu), however that number decreased to two at the start of winter term. Our resulting team, Adam Ruark and Vincent Nguyen, acted as the sole creators of this application. Adam's work consisted of establishing much of the back-end infrastructure and it's connections to the HyperRail, while Vincent's work focused on creating and integrating the user interface which would be the face of the project.

2 Requirements Document

2.1 Summarized Changes

Section	Original	New
Title Page	Three team members.	Yihong left the team. Updated abstract.
Limitations	HyperRail is limited to areas with low interference.	Personal computer must be near the HyperRail to connect and interact with it.
Database Requirements	Database requires safe and secure user login and will also store configurations.	Database does not require user login and is stored locally on the user's machine. Configurations are stored locally on the user's machine.
Reliability	No direct connection to the hardware.	Direct connection to the hardware is allowed.
Efficiency	The application should support multiple users.	Removed.
Integrity	The application should display or log what user triggered which action.	The application will notify the user whether an action was successful or failed and display a server log message.
Portability	Mobile devices and personal computers can connect access the web application.	Personal computers can host and access the web application and can interact with the hardware by connecting to its network.
Safety	The system will only allow registered users to log in to interact with the application.	Removed.
Gantt Chart	Chart displays planned progression.	Chart matches actual progression.

2.2 Introduction

2.2.1 Purpose

The purpose of this document is to provide an overview of the product to be delivered to the client by the end of Spring 2019. The client has listed a number of features, or project requirements, that they wish the capstone team to implement.

2.2.2 Scope

The hardware for the HyperRail project has already been decided upon and its code has been mostly developed. For this project, the hardware code needs to be optimized and a web-based graphical user interface and central server are needed to complete the software portion of the HyperRail application. The specified features are meant to allow a user to remotely communicate with a specified HyperRail system, which is handled and abstracted through the central server. The features have two levels of priority: required and convenient. Specifications that are required are needed for the system to be fully developed and deployed. Those that are convenient are stretch goals that are not required, but would be nice to have to make the system more robust.

2.3 Product Overview

2.3.1 Product Description

The HyperRail is a small railway where an automated environmental sensor package, which contains a variety of different sensors, can traverse through a space and collect information as it travels along the railway. Using the HyperRail application, the sensor package's settings can be customized and monitored. The application currently allows the user to specify the speed of the package, current length of the rail, and the size of the spool used for the motors. It also monitors the position of the sensor package along the railway by calculating the number of motor steps it has taken and updates the position display in real-time. However, it currently requires a direct connection, wired or wireless, to the sensor package itself, limiting the HyperRail's use.

2.3.2 User Characteristics

Users of the HyperRail will be using it to monitor environmental changes. They can monitor levels of carbon dioxide, moisture, light levels, and other variables in the environment depending on the sensors used in the package. No programming experience is required, but the user will need to know how to interact with computers as the application abstracts the programming required to automate the HyperRail system. The user can interpret the data collected in any manner they would like.

2.3.3 Limitations

The HyperRail is limited to local area connections. The personal computer must be within range to connect to the hardware and use the application. The HyperRail is limited to safe or contained environments. Because the HyperRail system utilizes a physical setup, the railway or the sensor package can be damaged by entities in the environment. The computer and sensor package also requires electricity to function. Therefore, the HyperRail is limited to areas with power.

2.4 Functional Requirements

2.4.1 Usability Requirements

This application must have a user interface that is intuitive to use. This means that the primary functionality of the app such as defining the parameters of the system's journey and uploading them to the robot should never be more than one click away. Secondary functionality such as saving and loading configurations can be a few clicks away. This is to provide a readable interface that is not cluttered.

The application must also provide feedback to the user when an action succeeds or fails. This can be done with a pop-up containing a status message for the last completed action.

2.4.2 Database Requirements

There are some attributes that must be stored in a database. First, all sensor data will be stored in a database that is locally stored on the user's machine. Configuration data will also be stored locally on the machine, but not in a database.

2.4.3 Code Quality

The underlying code that will be running this application must be maintainable for teams in the future to improve upon. This means that the code must be homogeneous and well documented. These qualities will allow the any portion of code to be understandable just from looking at its source and to have the reason for its existence documented.

2.4.4 Reliability

The application needs to allow the user to operate the HyperRail system remotely, using the custom-developed web interface to finish their specific request.

2.4.5 Efficiency

The application needs to be able to handle different commands from users accurately, quickly, and efficiently.

2.4.6 Integrity

Integrity for this system is defined by how well it can track the commands from the users then handle those commands successfully. This means that user actions must be accountable by displaying whether a user's action was successful or failed and creating a server log message.

2.4.7 Flexibility

The application should account for future changes. If functions are modified or added to the application, version control should be used to ensure that there is working code.

2.4.8 Portability

This application should be able to interact and configure new sensor packages without major changes to the software. The software itself should be portable in the sense that personal computers can host and access the web application and interact with a specified HyperRail system by connecting the the hardware's network.

2.4.9 Safety

This application will need to utilize a secure connection to ensure that all content is controlled and private.

2.4.10 Performance Requirements

The essential measurement standard for the performance requirement is if the application is able to satisfy all the requirements and include all of the functionality needed to be deployed. Although incorporating the stretch goals is not mandatory for the application's release, it will make the application more robust.

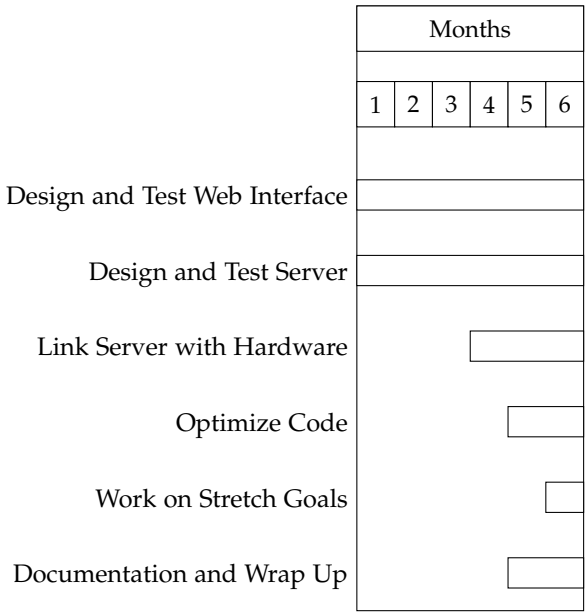
2.5 Stretch Goals

One stretch goal is to allow the sensor package to interact with existing actuators in the space. Because actuators are machines that perform simple movements on objects, such as valves or switches, this interaction will give the application additional capabilities depending on the application of the actuators.

Another stretch goal is the capability to configure a fleet of sensor systems to use the same configuration. This allows for quick configuration of a group of sensor packages that are intended for the same purpose.

Another feature that we would like to include is the capability to save and load configurations to allow users to share their settings across devices. This would make the application even more portable for users and allow the sensor system to be truly configured from anywhere. It would also allow the user to experiment with different configurations while preserving the old settings.

2.6 Gantt Chart



3 Design Document

3.1 Summarized Changes

Section	Original	New
Title Page	Three team members.	Yihong left the team.
Project Overview	Web application will be mobile friendly. Schedules can be made and runs can be scheduled.	Both removed.
User Interface	Planned views include the account, faculty, account settings, and sensor scheduling views.	These views are removed.
Server - Database	MongoDB will be used to store sensor data and configuration files.	InfluxDB will be used to store sensor data. Configuration files will be stored locally on the user's computer.
Functionality	The application will allow the user to configure schedule settings.	The application will allow the user to configure interval and sensor settings. Scheduling will be a stretch goal.

3.2 Introduction

3.2.1 Scope

The scope of this document includes the design decisions and rationale for each aspect of the HyperRail Application project. For each aspect, the desired functionality will be summarized and the methods, techniques, and/or tools that the team has decided to use to achieve said functionality will be discussed.

3.2.2 Purpose

This design document details the technologies to be used during the development of the HyperRail Application as well as the rationale behind each decision. After reading this document, a developer should know what components are needed for the project as well as the tools that can be used to implement the solution.

3.2.3 Intended Audience

This document is intended for the project client and the senior capstone advising team. Both the client and capstone advisors will use this document to examine and analyze the project team's design choices as well as monitor their progress during development.

3.3 Definitions

- HTML: A web markup language that can be used to define the structure of a web page.
- CSS: A web styling language.
- JavaScript: A web scripting language that can be used for the browser and on a server.
- NodeJS: A platform that can run JavaScript and host web pages on a server.

- WebStorm: a powerful IDE for modern JavaScript development. Besides client-side applications, WebStorm helps you develop server-side applications with Node.js, mobile apps with React Native or Cordova, and desktop apps with Electron.
- JSON: JavaScript Object Notation. This is a popular method of handling information with key-value pairs.
- MVC: Model - View - Controller. A traditional client/server representation. The View is the client's perspective of a system (usually a web page), the Controller is the server, and the model is the data passed between the two. It is the job of both the Controller and the View to interpret the data passed to it.
- API: Application program interface. This describes the functions a developer can use with a third-party tool

3.4 Project Overview

The HyperRail is a small railway where an automated environmental sensor package, which contains a variety of different sensors, can traverse through a space and collect information as it travels along the railway. Using the HyperRail application, the sensor package's settings can be customized and monitored. The application currently allows the user to specify the speed of the package, current length of the rail, and the size of the spool used for the motors. It also monitors the position of the sensor package along the railway and updates its location in real-time. Our proposed solution to this problem is to create a web application that communicates with a central server, which in turn communicates with the sensor system. This web application will be designed for personal computers. The web application will let the user set up intervals for the sensors to collect data and also display the current status of the sensor system. Once a user saves a configuration, these settings will be uploaded to a central server where it will be deployed to the corresponding sensor system. The data collected by the sensors will be uploaded to the central server and can be viewed by the web application.

3.5 Hardware

For the base of the environmental sensor package, the Adafruit Feather M0 board will be used as it has many general purpose input and output (GPIO) pins, useful for connecting multiple sensors, and it also has different variations that support various methods of communication, such as Wi-Fi. With 256KB of flash memory and 32KB of static random access memory, the board has lots of memory to work with as it processes configurations sent by the user.

Because the HyperRail Application is meant to be compatible with any sensor, there are no required sensors. Sensors that are used for the HyperRail will be connected to the GPIO pins on Adafruit Feather M0 board. The data collected by the sensors will be sent to the board, which will then transmit the information to the HyperRail web application so the user can interact with the data.

3.6 User Interface

The web application is a web interface for users to operate the sensor on the HyperRail system to collect corresponding data for their research. The client-side implementation of the web application user interface (UI) will be made using HTML, CSS, and JavaScript while NodeJS will be used for the server-side hosting. The current list of planned views are as follows:

- Home: A home page with general information about the HyperRail system and application, such as its documentation and features and contact information.

- Sensor Configurations: A page that requests the user to input and/or displays the current sensor configuration information, such as rail length, spool radius, and velocity. It will also display a list of saved configurations.
- Data information: The page shows the collected data from the sensor package on the HyperRail system.

3.7 Data Flow

The main work flow for our application is as follows:

1) The user opens up a browser and puts in the URL for the HyperRail application.
2) The server responds and returns a web page.
3) The user enters in some configuration values and clicks ”Run”.
4) The data gets sent back to the server and is processed.
5) Processing the information involves transforming it into values the Arduino running the HyperRail can understand.
6) The information is sent to the HyperRail and the system begins running.

There are many other work flows possible in our system, but the majority of the data passed around will go through similar steps. Overall, the key component here is that the central server processes the data before it is passed on to the HyperRail system. One other component not mentioned here is a database. Users will have the option to save and load their configurations, so data will need to be stored. This flow also relies on the central server to transform the data passed between the endpoints.

3.7.1 Server - User connection

There are two ways that data will be passed between the user and the server: static files and dynamic JSON payloads. We plan on using a traditional MVC method to handle the data passed around. This allows us to seamlessly manage information and it is accommodating to changes as new sensors or interface changes come about. On the initial connection between the client and the server, the first piece of information passed between the two is a web page. Our application will only have a couple different pages to serve so this interaction will not be discussed further.

The intensive portion of this connection will the be the JSON payloads passed between the two. These payloads will contain information such as HyperRail configurations, data collected from the HyperRail sensors, the current status of the HyperRail system, and more. Generally, the client will determine what action to perform by sending a request to a specific endpoint on the server and the JSON included in the request will contain the information specific to that unique request. For example, */config* with a payload containing containing the value *load* will tell the server that the user is requesting a configuration. The server will then send the requested configuration back to the client where it will be processed and displayed to the user.

3.7.2 Server - Database

The database will contain all the sensor data that is collected by the HyperRail system. The database will not store configurations or the status of the HyperRail system. The configurations will be stored locally on the user’s personal computer and the status will be resolved at run time.

The information that will get loaded into the database will come from two sources, the client or the HyperRail. The client will be sending and requesting HyperRail configurations from the database and the HyperRail will be uploading

collected data from each run. Since our server will be run off of NodeJS, we will be using InfluxDB as our database tool as our client requested. This is because InfluxDB provides a JavaScript API that allows a painless experience between the two technologies.

Internally, there will be a separation of concerns for what is stored in the database. The data collected by the HyperRail will be stored in a table, while the configurations uploaded by the user will be held locally on the computer. This should hopefully reduce the amount of processing needed when requesting information from the database.

3.7.3 Server - HyperRail

Lastly, the server must also communicate with the HyperRail. To simplify this, the connection will also be over HTTP (same as the server-client connection) to reduce the server overhead needed. The HyperRail system will be configured to listen to requests from the server and then execute them. Consequentially, the data sent over this connection will also be in the form of JSON. The HyperRail is tasked with processing the payload and then executing a run based on the information sent. Once a run is done, data is sent back to the server in the form of a JSON payload and is then stored in the database for use later. This connection will be abstracted away from the user and they should never have to worry about this process.

The information sent to the HyperRail will vary depending on the sensors available at the time and also the type of run the user wants to execute.

3.8 Functionality

The HyperRail application will allow the user to remotely interact with the physical HyperRail system. The application will allow the user to configure the sensor package's settings, such as the package's travel velocity, the motor's spool radius, the HyperRail length, interval information, and sensor settings, save them for future use, and upload them to the package itself. Configurations can be saved, reloaded, modified, and deleted in case the user wants to reuse certain settings or make room for new ones. Data collected by the sensors will be sent back to the web application where it will be displayed to the user.

Stretch goals for the application include run scheduling, automated data translation, fleet configurations, and actuator interactions. Run scheduling will allow the user to specify which sensors the package will use, where along the rail each sensor will collect data, and how frequent the package will traverse the rail. Automated data translation would examine the data collected and dynamically create actionable items or buttons for the HyperRail user to quickly analyze the information or take action. Fleet configurations would allow the user to configure a multitude of sensor packages at once, saving time and effort if they were used for the same tasks. Actuator interactions would allow the package to interact with existing actuators in the environment. Because actuators are used to perform simple motions, such as turning, pushing, or pulling, this would allow the HyperRail to interact with valves and pistons, which can help regulate the environment.

4 Adam's Tech Review

4.1 Introduction

The goal of our team is to develop an application for users to configure the HyperRail system from anywhere. The application should also be able to save and load configurations and delegate many systems at once. One of the key

components of this application is the web server. This server will act as the central hub for the connecting the user to the HyperRail system. This system will be in control of the user's data and verify that the configuration they create gets saved and sent to the robot succesfully. It will also monitor the status of each robot to ensure there is minimal downtime with the hardware. The realm of web hosting is a diverse field and there are many technologies out there for us to choose from. In this paper I will present three key questions regarding hosting a website and potential technologies to answer these problems.

4.2 How to Handle User Requests

The first task is determining how we need to route user requests and manage what the user can interact with. Below are some technologies that allow us to do this.

4.2.1 Django - Python Framework

Django is a widely used python library used to delegate a web server. This framework handles user requests, integration with databases, user authentication, and dynamic modeling of web pages before serving them. Django itself comes with a simple server, however this is not recommended and should be paired with a similar service such as Spawning. Python is a fantastic language for getting a lot done in a little amount of code and is human readable too. This advantage means that the people that carry on this project after us will have very little difficulty molding our code into whatever plans they have for the future and it should be picked up easily. The downside of this technology is that we still rely on another technology to create and run the server.

4.2.2 NodeJS - JavaScript Framework

NodeJS is a very popular server-side JavaScript framework used mainly to delegate requests sent to them. This JavaScript native platform allows us to utilize the countless JavaScript modules available on NPM and also lets us use the same language for both server-side and client-side code. It's fairly quick to get an application up and running and also quite versatile in what it can do. The advantage of this tool is that anyone taking on our work after us only has to be familiar with JavaScript and nothing else since both server-side and client-side code can be written in JavaScript. The downside of this is that the overall language has many caveats and the multitude of modules we'd use have their own patterns that would make understanding our code difficult at first glance. Overall this choice has a learning curve, but is very powerful once the user learns how to work with it.

4.2.3 PHP

PHP is super simple to get up and running and is great at serving out mostly static web pages very quickly. Creating a server in just as easy as in Javascript, but it also holds the advantage that any web pages served out can have inline PHP or JavaScript embedded in them. This presents more opportunities in which technology we can use once development begins.

4.3 How to Store User Data

The next issue we run into is how to store the user data. There are many solutions to this problem out there each with their own benefits and downsides. Below is a discussion of a few technologies we could use for this.

4.3.1 MongoDB

MongoDB is a management tool that is used to host and maintain a database. For a fee, MongoDB can host the databases for us and we just need to set up the architecture for it. Alternatively, we can download their engine and host a MongoDB instance on a local server instead. This has its own list of pros and cons but this means we don't have to pay a fee. MongoDB is also capable in integrating with all three frameworks discussed above making this a malleable solution.

4.3.2 Amazon DynamoDB

Amazon DynamoDB is Amazon's version of database hosting and is very similar to MongoDB. Amazon's databases must exist on their servers so there will be a third-party dependency. However, unlike MongoDB, they do offer a free tier to their service as long as the database stays under 25GB in size. The advantage of this is that this also allows seamless integration with any other AWS service we want to take advantage of and is scalable and stable. The disadvantage of this method is that we are reliant on a third-party dependency and introduces complexity to our architecture.

4.3.3 JSON or XML files

Lastly, due to the nature of this project, we might not even need a true database. The overall data we will be storing is minimal in size and can be stored in simple object files such as JSON, XML, or YAML. This means that we can maintain a simple architecture and have full control of our data. This also requires that we build that logic to maintain this and it may be better to use another solution.

4.4 Where to host the Server

Lastly, after creating a server and/or database, this needs to be hosted somewhere that can be accessed from the internet. We have a few different options of doing this

4.4.1 OSU/Local Servers

The first option is to host it locally. This means that we run the server off of an on-site central system that we maintain. The advantage of this is that we have full control of our dependencies and hardware and are not limited by third-party restrictions. However, it is another dependency we will have to monitor and if the service goes down, it's on us to figure out why and fix it. Overall this is a simple and cheap solution and may be a good stepping point before worrying about using a bigger technology.

4.4.2 Amazon EC2

Amazon EC2 offers server instances where we can host whatever we want. These also provide fantastic integration with their DynamoDB servers if we choose to go that route. Additionally, they offer monitoring tools and scalable systems if this project grows larger than anticipated. The downside of this tool is that Amazon does not offer this service for free (beyond a 12 month free period) and must be paid for. Additionally, this may be overkill for what we are attempting to accomplish.

4.4.3 HyperRail System

Lastly, we can "host" these directly on the Arduinos controlling the HyperRail systems. While this isn't a true web technology, it keeps each system independent and non-reliant to internet connectivity. The downside of this is that maintaining multiple systems at once will be difficult or near impossible and keeping all HyperRail systems in sync is

impossible. This will require an entirely different architecture than the one proposed by us, but it does carry a huge advantage in that internet connectivity is no longer a hard requirement. Due to the nature of where the HyperRail system will be use, this may be an idea we want to look into further.

4.5 Conclusion

Overall there are many different routes that we can take to handling the data that comes in and is ultimately used to run the HyperRail system. My suggestion for this application is a combination of NodeJS to create the server, JSON files to store user data, and our local servers to host the whole system. JSON files are native to JavaScript and should provide a homeogenous environment for the developer and our local servers are easier to maintain than relying on a third-party with data. There are other technologies out there that perform similar tasks and we may look into them down the road, but these offer much of the functionality we need and will most likely be used for this project.

5 VINCENT'S TECH REVIEW

5.1 Introduction

The technology review examines three components of the HyperRail project. For each component, three alternatives are explored for possible use in the project and each alternative will be evaluated based on their specifications.

5.2 Problem Description

The HyperRail is a small railway where an automated environmental sensor package can traverse along the railway through a space and collect information. Using the HyperRail application, the sensor package's settings can be customized and monitored. The application currently allows the user to specify the speed of the package, current length of the rail, and the size of the spool used for the motors. It also monitors the position of the sensor package and updates the position display in real-time. However, the application requires a direct connection to the sensor package, limiting the HyperRail's use.

5.3 Piece 1: Hardware Board

5.3.1 Overview of Criteria

The hardware board is the heart of the HyperRail environmental sensor package. The board is connected to all the sensors and is connected to the HyperRail application, allowing the board to communicate with the application. In addition to receiving commands from the application, the board will also send the package's current location and data collected from the sensors to the application for the user to see in real-time.

5.3.2 Adafruit Feather M0 Basic Proto

This is the hardware board that is currently selected for the HyperRail project. The board itself is small and lightweight, being 5.08 x 2.29 centimeters big and weighing 4.6 grams total. It has a total of 20 general purpose input and output (GPIO) pins as well as 6 analog inputs and 1 analog output. This allows a multitude of sensors to be connected without the use of an external board for additional connections. Despite the small size, it also has a built-in rechargeable battery that makes it useful for portable projects. An external power source, such as a battery, can be provided to continually charge the internal battery, but it is not required. The Feather M0 has 256 kilobytes (KB) of flash memory and 32 KB of

random access memory (RAM), but no electrically erasable programmable read-only memory (EEPROM). In addition to a direct serial connection, the board has several variations that enable communication with other devices through a Bluetooth or Wi-Fi connection.

5.3.3 Arduino MKR WiFi 1010

The Arduino MKR WiFi 1010 is the largest of the three alternatives, being 6.15 x 2.5 centimeters in size. It has fewer pins than the other two boards with 8 digital I/O pins, 7 analog input pins, and 1 analog output pin. This board has an internal lithium-polymer battery powering the board and can also be powered by an external source. Similar to the Adafruit board, it also has 256 KB of flash memory, 32 KB of static RAM, and no EEPROM. However, the MKR WiFi board can communicate with other devices through a direct or Wi-Fi connection and is cheaper than the Wi-Fi equivalent of the Adafruit Feather M0 board, meaning it will be easier for the board to interact with a web interface.

5.3.4 Arduino Micro

The Arduino Micro board is the smallest board of the three alternatives, being 4.8 x 1.77 centimeters big. The Arduino Micro also has 20 GPIO pins as the Adafruit Feather M0, but it does not have an internal battery, meaning the board requires an external power source. Because of its small and simple design, it can be easily placed on a breadboard to help with prototyping. The Arduino Micro has less memory than the other two boards, with 32 KB of memory, 2.5 KB of SRAM, and 1 KB of EEPROM. However, the board can only communicate with other devices only through a wireless or direct serial connection.

5.4 Piece 2: Hardware Communication Methods

5.4.1 Overview of Criteria

Hardware communication determines how the HyperRail environmental package will interact with the user. It determines how the package will receive configuration settings and commands from the user and how the package will send environmental data back to the user.

5.4.2 Wired Communication

Wired communication is the process of sending data over a wired communication channel. Typically with wired connections, the devices simply send bits sequentially through the cable. Data can also be simultaneous sent in both directions, to and from the board or devices, through the data plus and data minus wires. However, because the devices are directly connected to each other, the range is limited based on the length of the cable. For the HyperRail project, this direct cable connection may inhibit the HyperRail's use as it limits the environmental package's range and possibly its usability. The package would have to be connected to a computer at all times in order to send the data collected to the user. Depending on the HyperRail's physical setup, the long cables may be infeasible or too costly as the cable can get caught on objects or entities in the surrounding environment. However, for contained environments with a relatively short and linear rail, a cable connection could be useful for quickly transferring the data collected to the user.

5.4.3 Wi-Fi Communication

Wi-Fi communication is the process of sending data through the internet. Information is sent from one device to another using a router, which reads the information packet and routes it to an appropriate destination, either another router if

the target device is further away or the target device itself. Limitations to this type of communication is that it requires a stable internet connection. If the connection drops, data loss could occur and the HyperRail system would be offline, with the severity of both depending on the duration of the connection drop. However, because the proposed solution is to create a web application for the HyperRail, Wi-Fi communication would make it easier for the environmental sensor package to interact directly with the web application as it can send data directly to the web server for it to be available to the user.

5.4.4 Bluetooth Communication

Bluetooth is another form of wireless communication that uses low-power radio wave. Bluetooth uses weak radio signals which limits its communication range, but it reduces the chances of external interference with other Bluetooth devices. In the case of the HyperRail, the communication range becomes more important as the size of the railway increases, but for small HyperRail systems, the reliable and safe connection could be very useful especially in cluttered areas. It also avoids interference with other Bluetooth devices by rapidly changing its frequency. However, Bluetooth does not need a direct line of sight between the communication devices, so it can be useful in areas with many obstacles.

5.5 Piece 3: Hardware Sensors

The main purpose of the HyperRail is to make environmental data collection simpler. Therefore, environmental sensors are essential to the project so data collection can be automated, making environmental monitoring easier.

5.5.1 TMP36 - Temperature Sensor

Monitoring temperatures is vital in agriculture as certain crops or products require a certain temperature range for an optimal result. Also, as global warming becomes more prominent, monitoring temperatures becomes more important. The TMP36 temperature sensor is very small and cheap, costing $1.50 each. It is easy to setup, only having three pins: power, ground, and voltage out. The sensor outputs a voltage depending on the temperature it detects at a ratio of 10mV per 1°C and the sensor operates between -40°C and 125°C. At around 25°C, the temperature reading is with within 1°C of the true temperature, but near the ends of the operating range, the reading accuracy drops to being within 2°C of the true temperature. The temperature of most environments should be within this sensor's operating range, making this sensor useful for the HyperRail.

5.5.2 MG-811 CO2 Gas Sensor Module

Because global warming is caused by carbon dioxide (CO2) trapping heat in the Earth's atmosphere, monitoring CO2 levels becomes more important. The MG-811 sensor module is small and it seems a little pricey at around $52.95, but it is relatively cheap compared to other CO2 sensor modules. The module has a sensor jack and 4 pins which allow for simple and fast setup. The sensor itself can detect CO2 levels from between 350-10000 parts per million (ppm), which makes it very versatile, and outputs a voltage between 30 and 50mV depending on the CO2 level. The sensor's design allows it to essentially ignore other gases in the environment as well as the temperature and humidity.

5.5.3 DFR0027 - Ambient Light Sensor

Monitoring light level can be important for greenhouses as plants can require varying amounts of light to grow properly. The DFR0027 light sensor is also small and cheap, being 2.2 x 3 cm in size and costing $2.60. Like the temperature sensor, it is easy to setup as it only has three pins. The sensor can detect light levels from 1 to 6000 illuminance (lux) with a

response time of 15 microseconds, so it can detect abrupt changes in light levels. Although the lux of sunlight is greater than 6000, this sensor is designed to detect ambient light, so it can be useful for detecting small changes in darker environments.

5.6 Recommendations

Overall, I believe the Adafruit Feather M0 board is the most versatile and robust option. It has plenty of memory and plenty of GPIO pins, which means it can support many different sensors. It also has several variations that use different communication methods, so it can be used in many situations depending on the environment. For communication, I believe wired or Wi-Fi communication are good options. Wired communication has a fast data transmission and an easier implementation. It can be useful for linear railways because the cable will not get caught on obstacles. Wi-Fi communication is useful for larger areas that have internet because Wi-Fi connections can go through obstacles. It also makes interaction with a web interface much easier. For the sensors, although there are a vast amount of sensors, these three options perform their jobs at a relatively low price.

6 YIHONG'S TECH REVIEW

6.1 Problem description

The HyperRail is a small railway where an automated environmental sensor package, which contains a variety of different sensors, can traverse through a space and collect information as it travels along the railway. Using the HyperRail application, the sensor package's settings can be customized and monitored. The application currently allows the user to specify the speed of the package, current length of the rail, and the size of the spool used for the motors. It also monitors the position of the sensor package along the railway and updates its location in real-time. Our proposed solution to this problem is to create a web application that communicates with a central server, which in turn communicates with the sensor system. This web application will be mobile friendly to work on Android, iOS, and personal computers. The web application will let the user configure schedules for the sensors to run, set up intervals for the sensors to collect data, and also display the current status of the sensor system. Once a user saves a configuration, these settings will be uploaded to a central server where it will be deployed to the corresponding sensor system. Scheduling runs will be controlled and initiated by the central server by sending signals to the HyperRail sensor system with locations to go to, rate of travel, and any points of interest to monitor along the way. The data collected by the sensors will be uploaded to the central server and can be viewed by the web application.

6.2 Overview

This article introduces the detailed process about how to make a GUI for our project, there are 3 main parts, Wire-frame design, web solution and GUI testing. each of those technology points has the specific requirements for us to implement them in our project, we also find out some softwares to help us to achieve those functions. At the end, the user can use the GUI interface to operate the sensor package on the hyper-rail system to execute specific researching tasks without the mic-controller through our central web-server, our GUI interface will connect the web-sever to send the corresponding commands to the sensor. Therefore, we can achieve our goal, which is using the hyperrail system at anywhere.

6.3 Criteria

We will be able to use the GUI to let the sensor package on the hyper-rail system do execute the specific task as we command, the sensor package should be able to respond and execute the specific task with a short reaction time and high accuracy.

6.3.1 Wire-frame

First of all, we need to design our user interface, we need to have a user story and understand what the clients want, then we need a functional map, which means the sort our necessary functions according to our client's demands, then we need to design the UI flow and flow chart well to make sure the user interface can handle the commands efficient. At the end, we should have a prototype for us to test see if something needs to be improved or changed. here are some softwares that we might use for our UI design.

A. First one is about dynamic interaction, the app is called axure. this application is more focus on the dynamic pages and connect those pages to a dynamic effect

B. the second one is for static interaction, GUI design studio. this software is focus more on the page to page effect, its advantage is more concise layout and easier to use.

C. the third one is framer.js, this software is more complex than other two softwares that I mentioned, it provides creativity freedom and it has more platforms available for users. The code in Framer prototypes can be copied and used directly by developers to reproduce hi-fi interactivity patterns. it also have some disadvantages, such as the user needs to learn code, and import issues are also a trouble for some users.

At the end, we should have a hi-fi prototype of UI first, which fits all the requirement from the clients and then we will use the corresponding web language, such as css, html, node.js to handle the logical relationship in the GUI, and we will sent the prototype to our clients first to get some feedback or comments before we really implement it.

6.3.2 Web solution

For the specific web stuff that we are going to use with our user interface, I'm going to talk 3 main basic knowledge that we need to be able to design the UI.

A. Html is Hypertext Markup Language, a standardized system for tagging text files to achieve font, color, graphic, and hyperlink effects on World Wide Web pages. we can control the UI flow with Html, we can received the user's input to navigate the users to the page that they want, and we can upload images with html, what's more, we can do compute things through Html.

B. Css describes how HTML elements are to be displayed on screen, paper, or in other media. it saves a lot of work. It can control the layout of multiple web pages all at once. Compare to html, css is more focus on the layout of the interface page, such as color, font, template, spacing, etc.

C. Javascript is a scripting or programming language that allows you to implement complex things on web pages every time a web page does more than just sit there and display static information for you to look at displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. The common use for Javascrpit are image manipulation, from validation, and dynamic changes of content.

Besides that, we probally will use a lot of IDES to write the API for connecting our back end port, such as Webstorm, WebStorm is a powerful IDE for modern JavaScript development. Besides client-side applications, WebStorm helps you develop server-side applications with Node.js, mobile apps with React Native or Cordova, and desktop apps with

Electron. Also, this ide can be used to write node.js, css, html, and handle a lot of logic problems between front end and back end. Overall, no matter what a specific IDE we will use for our project, it should be able to handle with Node.js coding well, since all of our group members decide to use Node.js as a main tool to achieve the connection between between server-side and client-side.

6.3.3 GUI testing

After we have finished the UI prototype, we need to test the UI that we designed, I'm going to talk 3 main tools that can help us to do the test work in the future project.

A.Abbot is a java-based test framework, it can automatically to generate and verify java GUI, it can let the user to explore, operate the system easily. developer can through the script or compiling the code to use the Abbot framework.

B. EggPlant doesn't interact with the basic code, eggplant is more focus on those user interfaces with problems, those problems might be caused by FLash, java, HTML, net, etc. And the application can play back to the place that the interface has problems, so we can take a look at the specific crux of the problem, then fix it.

C. HP winRuner is a automatic GUI testing application, it tests the script and go back to check the interaction process in the UI, to find the disadvantages, the user can know which part in the UI flow didn't run as they expected. And we can do the conrresponding change to improved.

The reason why we want to the GUI testing is that there are many unpredictable bugs happen when we test the GUI, there are several bugs when we test the GUI, the fist one is Overlap, this means the context is overlapped some other contexts, the second one is text wrapping, one line sentence is getting split by two lines, the third one is lost shortcut, usually, every UI can be operated by keyboard, but if the user has no mouse, we definitely want to make sure those shortcut works fine on the UI. And those testing tools will help us to find the bug in our code.

6.4 User Interface layout

The web application is a web interface for users to operate the sensor on the hyperrail system to collect corresponding data for their research. The current implementation of the web application UI will be made by javaScript and NodeJS. The current list of planning views are as follows:

- Account: Provides views for account management, including login, registration, password recovery, login failure, email confirmation, other related account information.
- Facility: Provides views for facility staff to manage residents of their facility who own the Hyperrail system, and some related information about term of service.
- Home: A home page with general information about the Hyperrail system, such as documentation and contact information on the top bar.
- Setting: Provides additional views for account management beyond login, including password changes, binding other information to the account, logout, etc.
- Project: A place to save an unfinished research or create a new research, for example, the user can get the research process and the data from last time they use the Hyperrail system. Also, the user can create a search with different command parameters.
- Operating page : A page shows the correspond information that request the user to input, such as ”rail length” , ”spool radius” , ”velocity” , ”whatever user wants to test”, and user can change the input.

- Data information : the page shows the collected data from the sensor on the hyperrail system after the user entered the command parameters.

6.5 Discussion

According to the client's requirement, we need to create the user interface that allows user in different place to be able to operate the hyper-rail system remotely. First we need to design the high usability UI base on the research that we are going to do, then we need the make sure the html,javascript web stuff can work well with the central server that we are going to make. At the end, we need to use those test application to test whether our GUI is working correctly or not.

6.6 Conclusion

In this document, we introduced some basic processes about how to create a good UI for our project, and the specific softwares that we are going to use to help us to design, editing the user interface. Also the web stuff needs to cooperate with our back-end well to handle the commands that user's input and give the corresponding the action. At the end, we talked about the testing tools that we might going to use in our project, through those testing applications we can find out the ways to improve our UI's efficiency and user experience. Besides that, we concluded the specific reason why we choose those technologies and compares the advantages and disadvantages. Also, we camp up with a draft layout for our UI design, which we want to make sure our clients will accept our UI design. And my suggestion is that we can use the webstrom IDE to do the NodeJS to create the server, using farmer.js to make a UI prototype, doing the test abbot since most of our work is java-based, there are still other technologies out there can perform the similar tasks and we may need to use others, but I think we covered the most of technologies that we need to create a UI for our project.

6.7 Reference

[1]MDN web Docs. (2018). "What is Javascript."
[online] Available at: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript].

[2]blog.framer.com (2017). "Prototyping in Framer: Pros and Cons." Available at: https://blog.framer.com/framer-pros-and-cons-346778e091f8.

7 Weekly Blog Posts

7.1 Adam's Posts

- **Fall - Week 4**
  Progress: This week we've met up with our sponsor and he walked through the code with us. We also had our first meeting with our TA about the initial steps on the project.
  Problems: None
  Plans: Set up the Arduino environment on my computer and start picking around with the current code. Also designate one of us as our primary contact for our sponsor.
- **Fall - Week 5**
  Progress: Set up weekly meeting times with our sponsor
  Problems: None
  Plans: Need to figure out next steps. Will contact our sponsor for this.

- **Fall - Week 6**

  Progress: Worked on the requirements document and the tech review

  Problems: None

  Plans: Present the requirements doc to our sponsor for review.
- **Fall - Week 7**

  Progress: Worked on the tech report.

  Problems: None.

  Plans: Continue on class assignments. Show sponsor our plans.
- **Fall - Week 8**

  Progress: Talked to our client about the requirements document. He likes our direction.

  Problems: None

  Plans: Work on design document.
- **Fall - Week 9**

  Progress: worked on some proof of concept ideas

  Problems: none

  Next: design document work

- **Winter - Week 1**

  Progress: Nothing implemented over winter break. Some research into the hardware we're working with

  Problems: None at the moment. Just need to figure out our plan for the term

  Plans: Meet up with team members and discuss next steps.
- **Winter - Week 2**

  Progress: Had first meeting with our TA and contacted our client

  Problems: One member of our group (Yihong) has yet to talk to us and isn't registered for the class

  Plans: Figure out the teammate issue and meet with client to get hardware to work with
- **Winter - Week 3**

  Progress: Met up with client. Narrowed down some requirements and have a meeting set up next week to demo our progress

  Problems: None

  Plans: Integrate our pieces of the app together, prepare for demo next week.
- **Winter - Week 4**

  Progress: Met up with client. Narrowed down some requirements and have a meeting set up next week to demo our progress

  Problems: None

  Plans: Integrate our pieces of the app together, prepare for demo next week.
- **Winter - Week 5**

  Progress: Had poster critique session. Completed majority of back end and some integration with the front end

  Problems: Had a meeting to demo progress on Tuesday, snow cancelled that meeting. Need to reschedule.

  Plans: Reschedule meeting. Work on integrating robot to back end

- **Winter - Week 6**

  Progress: Started work on setting up Arduino code

  Problems: Have yet to get hands on hardware. Code compiles but is untested on the hardware

  Plans: Head in to lab on Monday to hopefully work with the hardware
- **Winter - Week 7**

  Progress: got hands on hardware. working on integrating with backend

  Problems: None

  Plans: continue integrating with hardware
- **Winter - Week 8**

  Progress: Received more legacy code from the client. working on integrating it with our code

  Problems: Legacy code is confusing and full of bad code. Refactoring needed to fit with our project

  Plans: Integrating our backend with the existing code. If time permits, refactor the code more.
- **Winter - Week 9**

  Progress: Breakthrough with WiFi communication. Should be able to hook up parts now

  Problems: Testing code may be difficult. Lots of moving parts and legacy code

  Plans: Continue integrating our code. Head in to the lab to test our code with the rest of the hardware.
- **Winter - Week 10**

  Progress: continued work on connecting bot to server

  Problems: might have to change our data schema a bit, haven't decided yet.

  Plans: Finish connecting the bot up and then go into the lab to test it

- **Spring - Week 1**

  Progress: Some code cleanup, validating everything works on the frontend and preparing the backend for testing

  Problems: None

  Plans: Head in to the lab to test the backend with the hardware
- **Spring - Week 2**

  Progress: Met with client to review our work. Got some feedback.

  Plans: Work on bug fixes and feedback our client gave us.

  Problems: Currently swamped with homework from other classes. Will be in a time crunch to finish everything.
- **Spring - Week 3**

  Progress: Code has been frozen. Finished revising various documents and sent them to client

  Problems: Client hasn't responded or verified our documents yet

  Plans: Remind our client that we need him to sign our documents.
- **Spring - Week 4**

  Progress: Worked on poster draft

  Problems: We sent our client our documents to verify, and he hasn't responded back to them yet.

  Plans: Verify our client's verification.
- **Spring - Week 5**

  Progress: Filling out forms in preparation for expo

Problems: None

Plans: Keep planning for expo

- **Spring - Week 6**

  Progress: Preparing materials for expo

  Problems: None

  Plans: Verify we have everything we need for expo and continue preparing

7.2 Vincent's Posts

- **Fall - Week 3** Progress: Ive been assigned the HyperRail application and have contacted my team and met up with my client. Weve discussed the project at a high-level, talking about the purpose of the project and goals for the year as well as introducing us to tools and topics that will be needed on the project. Ive also completed the Problem Statement rough draft assignment and submitted it.

  Problems: No problems have been encountered so far.

  Plans: I plan on setting up the software needed for the project as well as completing some tasks that the client has assigned to help us get practice interacting with hardware and the Arduino IDE.
- **Fall - Week 4** Progress: Weve met up with the client again to get the required programs setup as well as getting familiar with the code and the thought process behind it. Were also working on the final draft of the problem statement.

  Problems: No problems so far.

  Plans: I plan on getting started on the requirements document soon, contacting the team to gather our thoughts and compile our information on the specifications and desired results on the project.
- **Fall - Week 5** Progress: Weve just been grinding out these reports for the project.

  Problems: None

  Plans: We plan on meeting up with the client on week 6 to finalize the requirements for the project. We also plan on finishing up the requirements document and the tech review.
- **Fall - Week 6** Progress: Just more reports.

  Problems: Tired of reports.

  Plans: Finish all these reports, then try to meet with the client when we have time.
- **Fall - Week 7** Progress: Finalized tech review, had a weekly meeting with the client and received some feedback about current progress.

  Problems: None

  Plans: Meet with the client and finalize the requirements document before he leaves for a 10-day trip.
- **Fall - Week 8** Progress: Started looking into database structuring in case we decide to use a database for the project. Have also been looking into web development languages (HTML, PHP, CSS, JavaScript) and potential tools (JQuery, Bootstrap).

  Problems: None

  Plans: Discuss web application design and layout and try to create some diagrams.
- **Fall - Week 9** Progress: Looked more into web development languages and tools.

  Problems: None

Plans: Paper prototyping, have some preliminary diagrams and layouts.

- **Winter - Week 1** Progress: Began experimenting with HTML, jQuery, PureCSS, and Bootstrap
  Problems: None
  Plans: Will continue experimenting with the different tools and try to start making the web pages for the HyperRail App
- **Winter - Week 2** Progress: More experimentation with HTML, Javascript, and Bootstrap
  Problems: None
  Plans: Will start making basic web pages for the HyperRail App
- **Winter - Week 3** Progress: Developed the basic structure and functionality for the Home page for the HyperRail Application.
  Problems: Having trouble deciding on the UI design and filler text, such as coloring and information text, but will postpone until the applications essential functionality is properly working.
  Plans: Will finish up the Home page and move onto other pages such as the HyperRail Configuration page, View Data Collected page, or Account Management.
- **Winter - Week 4** Progress: Developing the basic structure and functionality for the HyperRail Configuration page and the Data Collection page. Finished the basic home page.
  Problems: Having trouble developing without any data to test the code. Also trying to learn how to interact with a Node JS server and how the Influx database works.
  Plans: Will continue working on the configuration and data page. Will try to get test data to test the pages and see if the front end is properly interacting with the back end.
- **Winter - Week 5** Progress: Tested the Data Collection Runs page and it correctly queries the database based on the search filters of run name, bot name, and sensor type. Made some progress on the configurations page.
  Problems: Not too sure on how to develop the HyperRail configurations page because the application is supposed to be able to configure when certain sensor types collect data.
  Plans: Will look more into hardware and researching how the Arduino code works.
- **Winter - Week 6** Progress: Worked on connecting the front end and back end more, specifically the Configurations/Scheduling page. Also, refreshed my memory on the current hardware code.
  Problems: The current hardware code doesnt have any code for the sensors which could be attached. This makes it difficult to make decisions on how to create configurations for the HyperRail, such as how the hardware will be able to tell if a certain sensor type is connected.
  Plans: Will create a general configurations/scheduling page and look into how the sensors will interact with the hardware.
- **Winter - Week 7** Progress: Still working on the configurations page. Will try to finish up this week and move onto improving the data display (runs) page and try to start the scheduling page.
  Problems: We still dont have access to the hardware sensor code, so were unable to modify it to send JSON data to and from our web application.
  Plans: Will continue forward with front-end development and try to get our hands on the sensor code from the lab.

- **Winter - Week 8** Progress: Finalized the configurations page and looking into improving the data display page.
  Problems: Scheduling will be very tricky as Arduino is single-threaded. Might have to implement this functionality as a stretch goal.
  Plans: We recently got access to sensor code and will now analyze it and adapt it to suit our needs. Will look into displaying the HyperRails status.
- **Winter - Week 9** Progress: Improved the UI to be more aesthetically pleasing and filled some content text. Also fixed empty query bug on runs page which broke the page.
  Problems: Been working on the front-end the entire time, so relearning the server-side JavaScript is slowing progress as I pick up more work on the back-end. The end of the term is coming up as well, so other projects and assignments are also slowing down progress.
  Plans: Will finish up the configs page server wise.
- **Winter - Week 10** Progress: Overhauled the configs page, but it is now fully functional on the front and back end. Fixed some bugs on the runs and configs page as well.
  Problems: None this time.
  Plans: Will update the progress report and work on the video for the project.

- **Spring - Week 1** Progress: Commenting and testing the code. Fixing a few bugs and adding quality of life features.
  Problems: Forgot that the data schema was changed, so the data display is currently broken.
  Plans: Going to have to overhaul the data display code to fix it and make it fit the new data schema. Finish application testing and documentation to ensure that everything works properly and consistently.
- **Spring - Week 2** Progress: Made changes to the UI and server as requested by the client and tested the application to ensure it works. Revised requirements and design documents.
  Problems: None this week.
  Plans: Get client approval on documentation changes.
- **Spring - Week 3** Progress: Finalized requirements and design documents revisions. Reviewed expo registration.
  Problems: Currently trying to get client verification on revisions.
  Plans: Get client approval on documentation changes. Review, update, and finish expo poster. Possibly do more clean up work on the project.
- **Spring - Week 4** Progress: Revised and submitted our poster for the Expo.
  Problems: Currently trying to get client verification on revisions.
  Plans: Get client approval on documentation changes.
- **Spring - Week 5** Progress: Client verification was received, expo poster was submitted for printing, and model release forms were turned in.
  Problems: None currently.
  Plans: Prepare for expo.
- **Spring - Week 6** Progress: Testing and commenting code to ensure it's ready for the expo.
  Problems: None currently.
  Plans: Prepare for expo.

COLLEGE OF ENGINEERING | Electrical Engineering and Computer Science | CS25

HyperRail Application

Automate Your Data Collection

Web Application Designed for Greenhouses

- It is developed using HTML, CSS, and JavaScript for the front-end web pages and NodeJS and InfluxDB for the server hosting and database.

Features

- With the application, a user can configure a HyperRail system, modifying the sensor package's velocity, data collection intervals, and utilized sensors.
- The HyperRail settings can be stored in configurations files which can also be edited or removed.
- The data collected will be stored in a database, where it can be viewed at any time through the application.

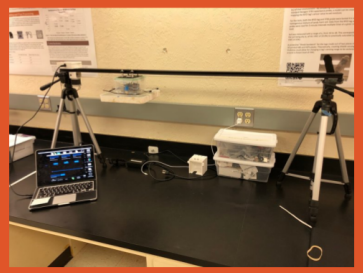


Figure 1. A small HyperRail setup.

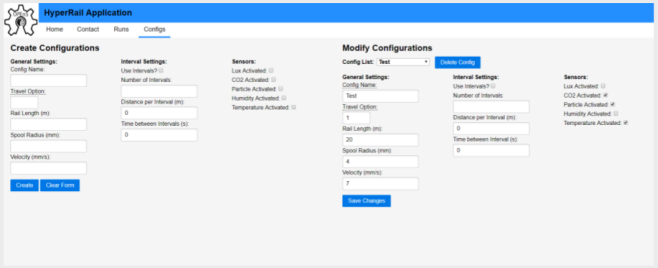


Figure 2. The Configurations Page allows users to create, modify, and delete configurations.

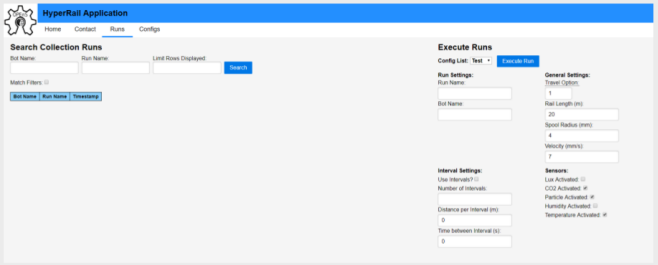


Figure 3. The Runs Page allows users to view collected data and execute data collection runs.

Project Architecture

- There are 4 major components: the server, the database, the user interface, and the bot.
- The user interface is a webpage that allows the user to control the HyperRail, save and load configurations, and browse the database of collected information.
- The server acts as a middle-man between the user interface, the database, and the bot.
- The database is powered by InfluxDB, which is a time-series data storage tool.
- The bot is powered by Arduino. This controls the movement of the bot, and also acts as a middle-man between the sensors and the server.

Project Background

- This application is part of a larger project. The HyperRail System is the sensor package on the monorail track that collects data. Our project, the application, is a controller for this system.
- The HyperRail System has the capability to travel along a rail system and take various measurements at various intervals.
- The HyperRail system has interchangeable sensors and can have variable rail length.
- More details can be found at www.open-sensing.org/hyper-rail

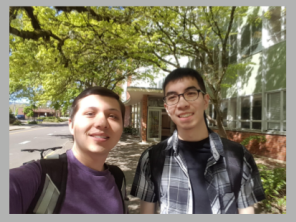
Why Use the HyperRail Application?

- Provides a portable and modular platform to simplify the collection of data.
- A robust web application that can run any HyperRail system connected to the network and enables automated data collection.
- Improve the reliability of collecting data and removes some hidden variables in that collection.
- Easily configure and deploy collection runs that automatically export the data to a database.
- A managed database ensures that all information collected is readily accessible and manageable to work with.

Team Members:
Adam Ruark -
Student
ruarka@oregonstate.edu

Vincent Nguyen -
Student
nguyvinc@oregonstate.edu

Client:
Chet Udell -
Lead at Oregon State's OPEnS Lab
udellc@oregonstate.edu



Oregon State University

9 PROJECT DOCUMENTATION

Note: This documentation was originally written in Markdown format, and we had difficulties importing it into this document. Consequentially, the embedded links are highlighted, but no longer work. Below is a list of every link in the document in the order they appear and their context. Additionally, the source code mentioned here is located at https://github.com/OPEnSLab-OSU/HyperRail

1) (Details on the HyperRail) http://www.open-sensing.org/hyper-rail/
2) (NodeJS can be downloaded here) https://nodejs.org/en/
3) (InfluxDB can be downloaded here) https://portal.influxdata.com/downloads/
4) (usage of the Adafruit Feather M0) https://www.adafruit.com/product/3010
5) (Arduino IDE can be downloaded here) https://www.arduino.cc/en/Main/Software
6) (Instructions can be found here) https://learn.adafruit.com/adafruit-feather-m0-wifi-atwinc1500/setup
7) (and here) https://learn.adafruit.com/adafruit-feather-m0-wifi-atwinc1500/using-with-arduino-ide
8) (ArduinoJson) https://github.com/bblanchon/ArduinoJson
9) (WiFi101) https://github.com/arduino-libraries/WiFi101
10) (RF24) https://github.com/nRF24/RF24
11) (OSC) https://github.com/CNMAT/OSC
12) (powered by Express) https://expressjs.com/

HyperRail Application and Hardware

Details on the HyperRail.

The HyperRail software is split between two packages: the HyperRail Application and the HyperRail Bot. The application is a NodeJS based server with a web-based user interface. This application is also powered by InfluxDB to store data collected from the bot. The HyperRail Bot is controlled by an Adafruit Feather M0 and runs on Arduino code.

Installation

The HyperRail application has several dependencies:

1. **NodeJS**
    - NPM and NodeJS can be downloaded here.
    - HyperRail server specific dependencies can be installed using `npm install` in the `./Code/server` directory.
2. **InfluxDB**
    - InfluxDB can be downloaded here.
3. **Arduino**
    - This project revolves around the usage of the Adafruit Feather M0.
    - Arduino IDE can be downloaded here
    - Once installed, the Arduino IDE must be configured for use with the Feather M0. Instructions can be found here and here.
    - Additionally, you will need to install several libraries. Most of these can be found through the Arduino IDE library manager under the tools menu in the Arduino IDE itself.
        - For the main HyperRail bot:
            - ArduinoJson
            - WiFi101
            - RF24
            - OSC

    One thing to note, there is an associated sensor hub that communicated with the bot, however this bot runs on outdated code that has since been updated by another team at the OPEnS lab. For the purposes of the Engineering Expo and the requirements of our application, we opted not to focus too much on integrating with this old code. Therefore, the current HyperRail bot code mocks a response from the sensors and sends that back to our server.

Running the HyperRail

NOTE: Be sure you have all dependencies installed before attempting to run the HyperRail.

HyperRail Bot

1. Open `./Code/HyperRail/HyperRail_with_Server/HyperRail_with_Server.ino` in the Arduino IDE. Compile and upload this code to your bot. At this point the bot can be unplugged from your computer and connected to a different power source.

HyperRail Application

1. Start the Influx database. If the executable is added to your system path, it can be started with `influxd`, otherwise find that executable and start it manually.
2. Start the HyperRail server with `npm start` from the `./Code/server` directory, verify the server starts successfully by the status messages printed to the console.
3. Connect to the HyperRail over WiFi. The SSID of the network is `HyperRail AP`.
4. In your browser, load up the user interface at the address `localhost:3000`.
5. From here, you are free to create/delete configurations, start executions of the HyperRail bot, and you can browse the database for results from each execution.

Development

Bot

Bot development is through the Arduino source code files located at `./Code/HyperRail/HyperRail_with_server`. The main file that runs the bot is `HyperRail_with_server.ino`

Application

Application development is through the source code files located at `./Code/server`. This is a NodeJS project, so dependencies can be installed with `npm install`. The server portion of this application is powered by Express and is defined in `server.js`. This application handles HTTP requests through routes defined in `src/routes`. These files are self-contained blocks of logic defining what happens when a user requests a resource at that route. There is also a `src/support` directory that contains blocks of logic that may need to be used in other files. Lastly, the `public` folder is for static files that get served out to the user. These include any html or client-side JavaScript that the user needs in their browser. Files here are sorted into subdirectories by their purpose.

10 Recommended Technical Resources

The primary resources we used to work on this project are included in project documentation in the section above. Additionally, although our knowledge on the following topics were gained through classes offered at OSU, we also recommend the documentation on the below topics as a crash course in some of the concepts utilized in this project

- jQuery (https://api.jquery.com/)
- HTTP Status Codes (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)
- Arduino Documentation (https://www.arduino.cc/en/Main/Docs)
- MVC (https://blog.codinghorror.com/understanding-model-view-controller/)

11 Conclusions and Reflections

11.1 Adam

Overall, this project provided a significant amount of experience of creating a product from scratch. Before this project, I have had technical experience with web servers and NodeJS in particular, however this project also offered a lot of insight into *Internet of Things* techniques. Specifically, working with Arduino, while arduous, gave me new insight on how to work with limited hardware. In a non-technical scope, I learned a lot on how to work with a client to build requirements on a project and how to collaborate on something like this with a team. Unlike group projects in other classes, the requirements aren't handed to us and a lot of it must be solved on the go. As such, there was a lot of discussion on the team about what would be best for each situation we ran into. Frequently there wouldn't be a clear-cut best solution and sometimes the solution we chose turned out not to be the right one. The biggest take-away from this was that the majority of a project isn't necessarily just writing code, it's about determining what code to write.

If I had to do this all over again, there would be a few keys steps I would change. First off, we had a slow start in the beginning due to some initially vague requirements given by our client. Consequentially, we brainstormed a multitude of solutions to these requirements, only to have most of our ideas snuffed out at the next requirements meeting. In the future, I would press harder for my requirements before delving too deep into a project; this saves time and effort in the long run. Second, I would narrow down the scope of the project immensely. Our initial idea had a much grander scope than what we have documented, however after we started working on it, problems we didn't even think of kept popping up and we had to cut down our scope. Lastly, there were a few requirements that our client set that had to be met that I personally didn't feel like they made sense. These weren't necessarily features that needed to be implemented, but requirements about what tools we had to use to implement these features. In the future, I would present my case to use a different tool, and at that point if my client still insists on their chosen tool, I'd implement that without question.

11.2 Vincent

This project taught me a lot about Arduino, about how to program an Arduino and how it can or can't be used, despite being given the code by our client. I'm pretty familiar with front-end development using HTML, CSS, and Javascript, but the project also refreshed my knowledge about server-side development and how they both interact.

For the non-technical aspect, this project helped me learn how to interact with clients, gathering information and requirements for the project. It also gave me insight on researching, looking up potential solutions or components that could be used for the project, and concurrent development with interdependent parts. I learned that it can be difficult

to gauge and appropriately scale the initial scope of a project as well as maintaining consistent communication between team members and project participants.

If I were to redo the project, I would be more vocal and ask more questions about the rationale behind certain requirements or design decisions that our client made. This would've sped up the development process as our goals would've been more concrete. I would also try to debate more with our client. Our team took and tried to implement all the requirements and design choices set by the client without trying to present or defend other alternatives that would've made the project easier or more robust.

12 APPENDIX 1: ESSENTIAL CODE

Listing 1. The HTML structure of the execute run form on the Runs page.

```
<div id='executeRunSide' class='pure-u-1 pure-u-xl-8-24'>
   <h2 class='header'> Execute Runs </h2>
  <h4 class='smallHeader'> Config List:
    <select id='configList'>
      <!-- Append Config List here -->
    </select>
    <button id='executeRun' class='pure-button pure-button-primary'>
       Execute Run
    </button>
  </h4>
  <form class='pure-form pure-form-stacked pure-g'>
    <fieldset>
      <div class='pure-u-1 pure-u-sm-1-2 pushTop'>
        <b> Run Settings:</b>
        <label for='runNameInput'>Run Name:</label>
        <input id='runNameInput' type='text'/>

        <label for='botNameInput'>Bot Name:</label>
        <input id='botNameInput' type='text'/>
      </div>

      <div class='pure-u-1 pure-u-sm-1-2 pushTop'>
        <b> General Settings:</b>
        <label for='modifyOption' class='tooltip'>
          <span class='tooltiplabel'>Travel Option:</span>
          <span class='tooltiptext'>
            1 = regular travel<br>
            2 = travel to end of rail<br>
            3 = travel to start of rail<br>
            4 = travel backwards<br>
            5 = travel forwards
          </span>
        </label>
```

```
  <input id='modifyOption' type='number' min='1' max='5'/>

  <label for='modifyRailLength'>Rail Length (m):</label>
  <input id='modifyRailLength' type='number' min='0' step='0.01'/>

  <label for='modifySpoolRadius'>Spool Radius (mm):</label>
  <input id='modifySpoolRadius' type='number' min='0'/>

  <label for='modifyVelocity'>Velocity (mm/s):</label>
  <input id='modifyVelocity' type='number' min='0'/>
</div>

<div class='pure-u-1 pure-u-sm-1-2 pushTop'>
  <b> Interval Settings:</b>
  <label for='modifyIntervalFlag' class='pure-checkbox'>
     Use Intervals?
    <input id='modifyIntervalFlag' type='checkbox'/>
  </label>

  <label for='modifyStops'>Number of Intervals:</label>
  <input id='modifyStops' type='number' min='0'/>

  <label for='modifyIntervalDistance'>Distance per Interval (m):
  </label>
  <input id='modifyIntervalDistance' type='number' min='0'/>

  <label for='modifyTimeInterval'>Time between Interval (s):
  </label>
  <input id='modifyTimeInterval' type='number' min='0'/>
</div>

<div class='pure-u-1 pure-u-sm-1-2 pushTop'>
  <b> Sensors:</b>
  <label for='modifyLuxActivated' class='pure-checkbox'>
     Lux Activated:
    <input id='modifyLuxActivated' type='checkbox'/>
  </label>

  <label for='modifyCo2Activated' class='pure-checkbox'>
     CO2 Activated:
    <input id='modifyCo2Activated' type='checkbox'/>
  </label>

  <label for='modifyParticleActivated' class='pure-checkbox'>
     Particle Activated:
```

```
            <input id='modifyParticleActivated' type='checkbox'/>
          </label>

          <label for='modifyHumidityActivated' class='pure-checkbox'>
             Humidity Activated:
            <input id='modifyHumidityActivated' type='checkbox'/>
          </label>

          <label for='modifyTemperatureActivated' class='pure-checkbox'>
             Temperature Activated:
            <input id='modifyTemperatureActivated' type='checkbox'/>
          </label>
        </div>
      </fieldset>
    </form>
</div>
```

Listing 2. The client-side JavaScript code to execute a run from the Runs page.

```
function executeRun() {
  var err = ""; //Strings to store error messages
  var optErr = "";
  var numErr = "";


  if($.trim($("#runNameInput").val()) == "")
    err += "Run Name, ";
  if($.trim($("#botNameInput").val()) == "")
    err += "Bot Name, ";

  if($("#modifyIntervalFlag").prop("checked") == false) //Convert Interval flag checkbox
      status to integer
    var intervalFlag = 0;
  else
    var intervalFlag = 1;

  if($("#modifyOption").val() == "")     //Check if the option number is empty
    err += "Option, ";
  else if ($("#modifyOption").val() < 1 || $("#modifyOption").val() > 5) //If not empty,
      check if it is between 1 and 5
    optErr += "Option must be an integer between 1 and 5.\n";

  if($("#modifyRailLength").val() == "")    //Check if the Rail Length is empty
    err += "Rail Length, ";
  else if ($("#modifyRailLength").val() <= 0)   //If not empty, check if it is 0 or less
    numErr += "Rail Length, ";
```

```
if($("#modifySpoolRadius").val() == "")    //Check if the Spool Radius is empty
  err += "Spool Radius, ";
else if ($("#modifySpoolRadius").val() <= 0)  //If not empty, check if it is 0 or less
  numErr += "Spool Radius, ";

if($("#modifyVelocity").val() == "")    //Check if the Velocity is empty
  err += "Velocity, ";
else if ($("#modifyVelocity").val() <= 0)   //If not empty, check if it is 0 or less
  numErr += "Velocity, ";

if(intervalFlag == 1){ //Interval settings only matter if intervals are used
  if($("#modifyStops").val() == "")    //Check if the Number of Intervals is empty
    err += "Number of Intervals, ";
  else if ($("#modifyStops").val() <= 0)   //If not empty, check if it is 0 or less
    numErr += "Number of Intervals, ";

  if($("#modifyIntervalDistance").val() == "") //Check if the Distance per Intervals is
      empty
    err += "Distance per Interval, ";
  else if ($("#modifyIntervalDistance").val() <= 0) //If not empty, check if it is 0 or
      less
    numErr += "Distance per Interval, ";

  if($("#modifyTimeInterval").val() == "")  //Check if the Time between Intervals is empty
    err += "Time between Intervals, ";
  else if ($("#modifyTimeInterval").val() <= 0) //If not empty, check if it is 0 or less
    numErr += "Time between Intervals, ";
}

if(err != "" || optErr != "" || numErr != "") {  //If there were any errors specified
  if(err != ""){
    err = err.slice(0, -2) + " must be specified."; //Format the error
    if(numErr != "")
      err += "\n";
  }
  if(numErr != "")
    numErr = numErr.slice(0, -2) + " must be greater than 0."; //Format number errors
  alert(optErr + err + numErr);  //Alert the user
  return;
}

if($("#modifyLuxActivated").prop("checked") == false) //Convert Lux Sensor flag checkbox
    status to integer
  var luxActivated = 0;
```

```
else
  var luxActivated = 1;

if($("#modifyCo2Activated").prop("checked") == false) //Convert CO2 Sensor flag checkbox
    status to integer
  var co2Activated = 0;
else
  var co2Activated = 1;

if($("#modifyParticleActivated").prop("checked") == false) //Convert Particle Sensor flag
    checkbox status to integer
  var particleActivated = 0;
else
  var particleActivated = 1;

if($("#modifyHumidityActivated").prop("checked") == false) //Convert Humidity Sensor flag
    checkbox status to integer
  var humidityActivated = 0;
else
  var humidityActivated = 1;

if($("#modifyTemperatureActivated").prop("checked") == false) //Convert Temperature Sensor
    flag checkbox status to integer
  var temperatureActivated = 0;
else
  var temperatureActivated = 1;

$.ajax({
  type: "POST",
  url: "/bots/execute",
  contentType: "application/json",
  data: JSON.stringify({
    runName: $("#runNameInput").val(),
    botName: $("#botNameInput").val(),
    option: parseInt($("#modifyOption").val()),
    railLength: parseFloat($("#modifyRailLength").val()),
    spoolRadius: parseFloat($("#modifySpoolRadius").val()),
    velocity: parseFloat($("#modifyVelocity").val()),
    intervalFlag: intervalFlag,
    intervalDistance: parseFloat($("#modifyIntervalDistance").val()),
    stops: parseInt($("#modifyStops").val()),
    luxActivated: luxActivated,
    co2Activated: co2Activated,
    particleActivated: particleActivated,
    humidityActivated: humidityActivated,
```

```
        temperatureActivated: temperatureActivated,
        timeInterval: parseInt($("#modifyTimeInterval").val())
      }),
      success: function(){ //Once the server is finished loading the config file, fill in the
          fields
        alert("Run execution successful.");
      },
      dataType: "json"
    }).fail(function(){
      alert("Run execution failed. Please check bot connection.");
    });
}
```

Listing 3. The server-side JavaScript code to execute a run from the Runs page.

```
router.post('/execute', (req, res) => {
    let config = req.body;
    const address = ip.address();

    /*
    Config missing total steps, delaytime, intervalSteps
    CALCULATIONS TAKEN FROM THE PROCESSING CODE

    Delay time = ((60)/(RPM * steps_per_revolution ) * pow(10, 6))/2;
    steps_per_revolution = 6180
    RPM = (velocity * 60) / (2 * 3.1415926535 * spoolRadius(mm))

    Total Steps = (steps_per_revolution * path_length (mm)) / (2 * 3.1415926535 * spoolRadius)
    Interval Steps = (steps_per_revolution * interval_length (mm)) / (2 * 3.1415926535 *
        spoolRadius)
    */
    let steps_per_revolution = 6180;
    let RPM = (config.velocity * 60) / (2 * 3.1415926535 * parseFloat(config.spoolRadius));
    let delayTime = (60 / (RPM * steps_per_revolution) * Math.pow(10, 6)) / 2;

    let totalSteps = (steps_per_revolution * (parseFloat(config.railLength) * 1000)) / (2 *
        3.1415926535 * parseFloat(config.spoolRadius));
    let intervalSteps = (steps_per_revolution * (parseFloat(config.intervalDistance) * 1000))
        / (2 * 3.1415926535 * parseFloat(config.spoolRadius));

    config.delayTime = parseInt(delayTime);
    config.totalSteps = parseInt(totalSteps);
    config.intervalSteps = parseInt(intervalSteps);

    logger.ok("Sent to hardware:\n" + JSON.stringify(config));
```

```
    if(!address) {
        logger.error('IP Address for the host could not be found');
        const status = 500;
        const msg = logger.buildPayload(logger.level.ERROR, 'Could not find host IP Address
            programmatically');
        res.status(status).send(msg);
    } else {
        config.ipAddress = address;
        axios.post(`http://${botAddress}/execute`, config)
            .then((botRes) => {
                const str = 'Config uploaded to bot, executing...';
                logger.ok(str);
                let status, msg;
                if(botRes.data.Status === "Received") {
                    status = 200;
                    msg = logger.buildPayload(logger.level.OK, str);
                } else {
                    status = 500;
                    msg = logger.buildPayload(logger.level.ERROR, 'Connected to bot, but upload
                        failed');
                }
                res.status(status).send(msg);
            })
            .catch((err) => {
                const str = 'Error uploading config to bot';
                logger.error(`${str}: ${err}`);
                const status = 500;
                const msg = logger.buildPayload(logger.level.ERROR, str);

                res.status(status).send(msg);
            });
    }
});
```

Listing 4. The server-side JavaScript code to connect to the Influx database.

```
exports.connect = () => {
    return new Promise((resolve, reject) => {
        if(client == null) {
            client = new Influx.InfluxDB(metadata);
            client.createDatabase(dbName)
                .then(() => resolve(`${dbUrl}:${dbPort}`))
                .catch((err) => reject(err));
        } else {
            reject('DB connection already established');
        }
```

```
    });
};
```

Listing 5. The server-side JavaScript code to collect data from hardware and store it in the Influx database.

```
//Creating a data entry in the runs database
router.post('/create', (req, res) => {
    const client = db.get();

    // Format data for database
    const formatData = [{
        fields: {
            value: JSON.stringify(req.body.data)
        },
        tags: {
            botName: req.body.botName,
            runName: req.body.runName
        },
        timestamp: new Date()
    }];

    // Save live data
    saveLiveData(req.body.data);

    // Write to database
    client.writeMeasurement(measure, formatData)
        .then(() => {
            logger.ok("Data uploaded to database");
            const msg = logger.buildPayload(logger.level.OK, 'Data uploaded');
            const status = 201;
            res.status(status).send(msg);
        })
        .catch((err) => {
            logger.error(err);
            const msg = logger.buildPayload(logger.level.ERROR, 'Error uploading');
            const status = 500;
            res.status(status).send(msg);
        });
});
```