CS Capstone Progress Report

March 17, 2019

HyperRail App

Prepared for

OPENS Lab

Chet Udell

Prepared by

Group 25

Vincent Nguyen
Adam Ruark

Abstract

This document discusses the progress and current standings on our work in developing the HyperRail application during Winter term. It also details issues that have slowed our progress as well as what is to be completed to be done with the project.

CONTENTS

1 Overview

The HyperRail system is a set of sensors attached to a robot that travels up and down a rail. While the robot travels, the sensors periodically take measurements, such as temperature or humidity, and records them. The hardware for this system has already been chosen and implemented, leaving us with the purpose of this project: to create a web-based application to improve the usability of this system, enhance existing features, and to add new features that improve the usability of the system. Some goals of this application are to allow users to view collected data from runs, create configurations for future runs, schedule runs for the HyperRail system, and to provide more clarity on the status of the HyperRail while it is running.

2 Current Progress

The HyperRail application has 3 major components: the front-end user interface, the back-end server, and the HyperRail bot.

2.1 User Interface

The user interface is moderately fleshed out with basic styling and structure and several connections to the server set up. It currently has 2 static pages, Home and Contact, and 2 pages, Runs and Configs, with dynamic content. The Home page displays a short and simple description about the HyperRail application and its use along with an image of the full setup. The Contact page currently has a link to the Openly Published Environmental Sensing (OPEnS) Lab website.

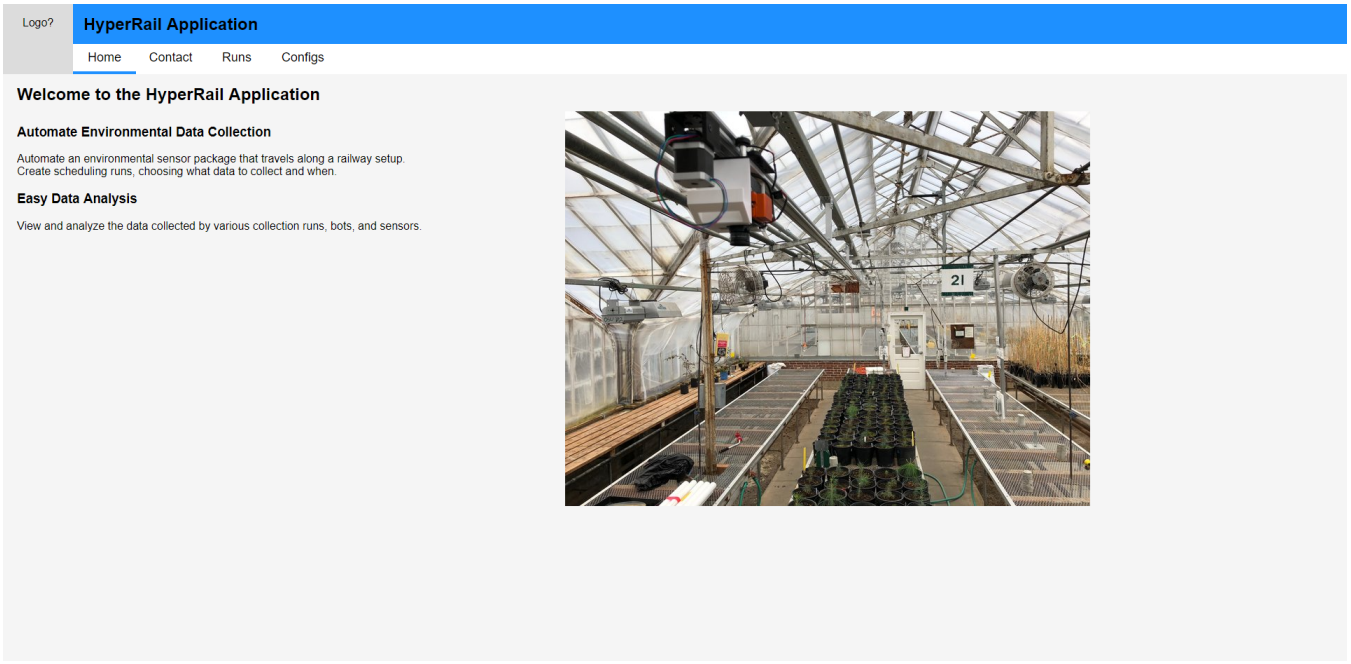

Fig. 1: The Home Page

The Runs page allows the user to look at the data collected. The user can enter a filter string for the bot name, run name, and sensor type and a number in the Limit Rows Displayed field and either press the Search button on the page or the Enter button on their keyboard to query the Influx database. There is also a checkbox that tells the server whether or not to search for the exact filter string in the database.

The server will handle the database querying and will return any data rows in the database which contains all the specified filter strings in their respective fields. If no filter strings were specified, the server will simply get all the data in the database. If no number was entered in the Limit Rows Displayed field, then all the data entries which match the filters will be displayed in the table. Otherwise, only the specified number of entries will be displayed. For example, in figure 2, all rows in the database which contained the string ”run1” in the Run Name column and the string ”the” in the Sensor Type column are displayed in the table.

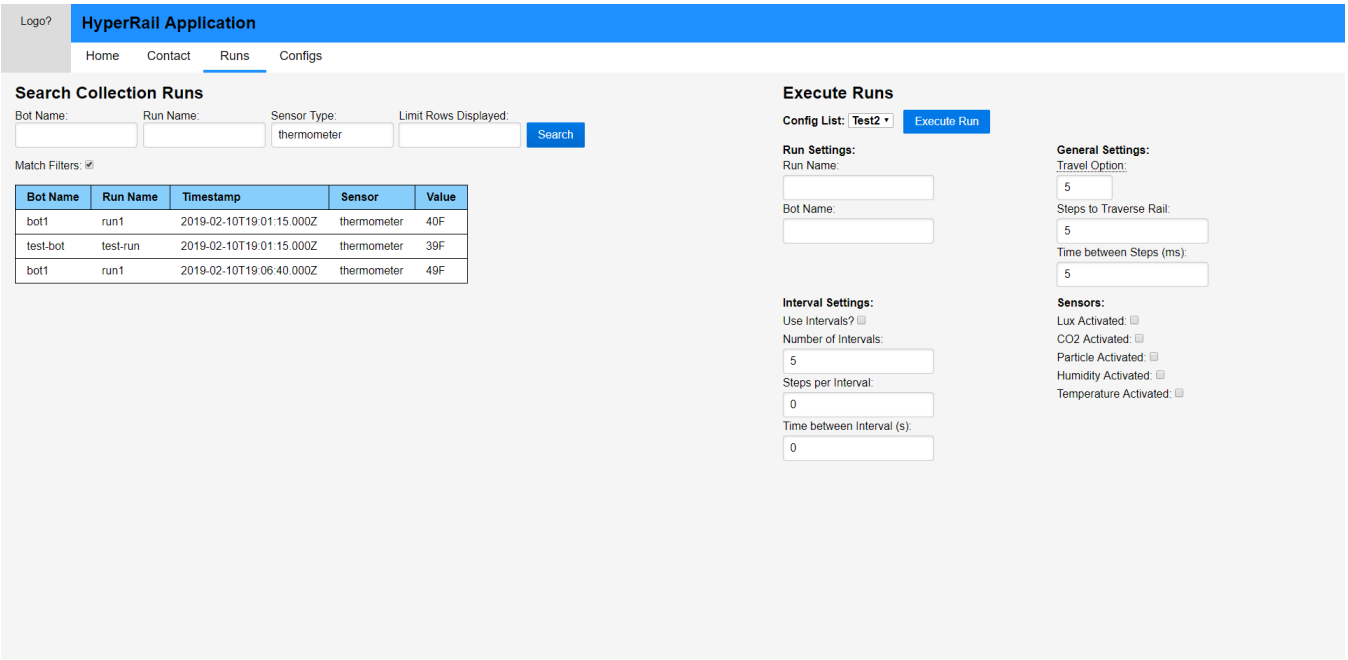

Fig. 2: The Runs Page

The Configs page allows the user to create and edit configurations for the HyperRail bot. Configurations have general settings, interval settings, and sensor activation settings. General settings include the configuration name, travel option number with an on-hover tooltip to explain each option, steps needed to traverse the rail, and the time between steps. Interval settings include a flag to use intervals or not, the number of intervals wanted, steps to take per interval, and the time between intervals. Sensor activation settings let the bot know whether or not to use lux, carbon dioxide, particle, humidity, and temperature sensors.

The form on the left side creates configurations and the form on the right side modifies configurations. Whenever a configuration from the Config list is selected, all of the fields are updated to match its configuration contents. The only restriction on configuration creation and modifications is the config name must be different from all other configurations including itself. This was to avoid accidental overwrites.

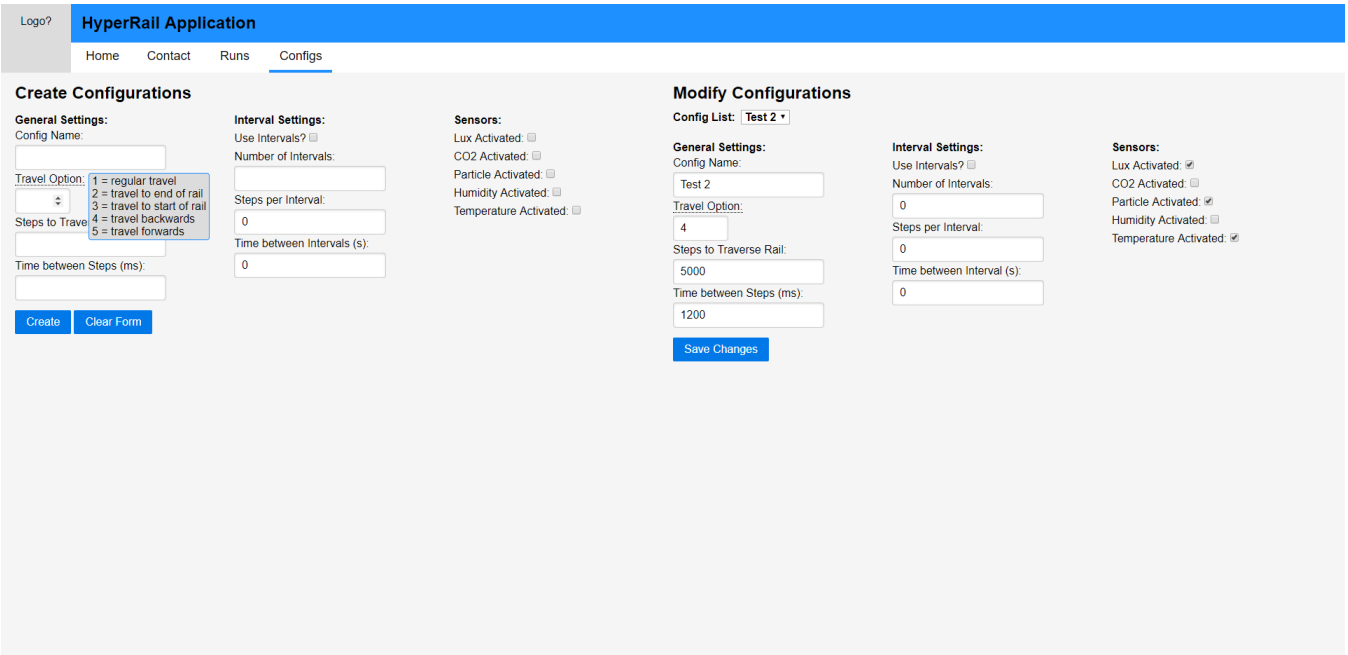

Fig. 3: The Configs Page

2.2 Server

Currently, the server is the most mature component. It has a fully integrated connection to a local InfluxDB instance and several endpoints to communicate with the user interface and the HyperRail hardware. For the server, there are three primary routes for the application: configs, runs, and bots.

The configs route contains all of the creates, reads, updates, and deletions (CRUD) for the HyperRail configurations page. Currently, this route allows the user to create a configuration, read and edit an existing configuration while deleting the old one, and list all the current configurations. These URLs take the JSON object given by their respective AJAX request and perform their respective action. For example, in figure 4, the server uses the config name entered as the name of the new JSON configuration file. If the name isn't taken, a file is created and the JSON configuration data is saved within the newly created file.

The runs route contains the CRUD for the collection runs page. Currently, this route allows for the insertion of data entries and the querying of existing data entries. For the data querying, the user is allowed to filter the data based on three inputs, the bot name, run name, and sensor type. The server code will use the non-empty search filters to query the Influx database and return the data tuples which contain the entered filter strings. If no search filters were entered, then all the data entries are returned.

The bots route contains all the interactions with the hardware. Currently, the only two actions are to check the status of the HyperRail bot, this is a work in progress, and to execute a data collection run. The collection run execution attempts to connect to the bot through a WiFi connection, then takes a configuration file and uploads it to the bot as a JSON object which will start a collection run.

The server also has 4 support routes: db, IP, logger, and monitor. The db route connects the node server to the Influx database. The IP route provide a method for the server to read an IP address. The logger route provides server logging support to help with troubleshooting, error logging, or action logging. The monitor route, a work in progress, allows

```
// Create a config file for the HyperRail
router.post('/create', (req, res) => {
    const fileName = `${req.body.configName}.json`;
    const data = req.body.data;

    const filePath = path.join(configDir, fileName);

    ensureDirs();
    logger.ok(`Created config: ${fileName}`);

    // Check if the config name exists
    if(fs.existsSync(filePath)) {
        const msg = logger.buildPayload(logger.level.ERROR, 'Config already exists, update instead');
        const status = 403;
        res.status(status).send(msg);
    } else {
        // If the config doesn't exist, create a new file
        fs.writeFile(filePath, JSON.stringify(data), (err) => {
            let msg, status;
            if(err) {
                logger.error(err);

                msg = logger.buildPayload(logger.level.ERROR, 'Error writing config to file');
                status = 500;
            } else {
                msg = logger.buildPayload(logger.level.OK, 'Config created');
                status = 201;
            }
            res.status(status).send(msg);
        });
    }
});
```

Fig. 4: Config Creation URL

```
// Params: Any tag defined in the db schema
// Search the database for data entries
router.get('/search', (req, res) => {
    const client = db.get();
    let dbQuery = `SELECT * FROM ${measure}`;
    let empty = true;

    dbQuery += ' WHERE ';
    // Build query from parameters (if parameters exist)
    for(let key in req.query) {
        if(req.query[key] != '') {
            empty = false;
            // dbQuery += `"${key}"='${req.query[key]}' AND `;
            // Equivalent to a LIKE query
            dbQuery += `"${key}" =~ /${req.query[key]}/ AND `;
        }
    }
    // Trim extra ' AND ' or ' WHERE ' at end of string if parameters exist or not
    if(!empty) {
        dbQuery = dbQuery.substring(0, dbQuery.length - 5);
    } else {
        dbQuery = dbQuery.substring(0, dbQuery.length - 7);
    }

    // Run query
    client.query(dbQuery)
        .then((result) => {
            res.json(result);
        })
        .catch((err) => {
            logger.error(err);

            const msg = logger.buildPayload(logger.level.ERROR, 'Error reading database');
            const status = 500;
            res.status(status).send(msg);
        });
});
```

Fig. 5: Query Database URL

the server to add and remove connected bots as well as checking and updating their current status.

2.3 HyperRail Bot

The overall data flow between the HyperRail bot and the HyperRail server is a bit odd. The bot acts as a WiFi hot spot for the user's computer to connect to and communicate with. However, due to the limitations of the hardware and available libraries, the bot cannot realistically maintain a constant connection with the server, and doesn't care who or what is connected to it. Therefore, we've implemented a basic handshake that tells the bot who is calling it and where to send the data back to. In this call, the user will also provide a configuration which the bot will execute indefinitely. During each step of the process, the bot will transmit sensor data back to the server (using the aforementioned metadata provided in the handshake) and the server will process it.

Of these steps, we have currently implemented the handshake process and are actively working on configuring how the bot will send data back to the server. Additionally, we are also working on refactoring some of the existing code to better fit with our data model and to help make the code more maintainable. With these changes we are getting rid of using Google Sheets as our data store and also removing an intermediate hub used to upload to Google Sheets. This is also a work in progress, but should help reduce the complexity of the project.

3 What's Needed

We still need to complete the interfacing with the HyperRail bot. The lab has provided code to make the hardware package travel along the rail and there is code to connect to the sensors and upload data collected to Google Sheets, but we very recently got access to the existing hardware and code running the HyperRail system, and now we need to modify it to work with our application. Once we connect our server to the bot, and modify it to upload JSON data to our server, the application will be ready for use.

4 Problems

One of our team members had to leave our capstone team, leaving us with a total of two members. This has slowed down our progression as our manpower has been reduced. Even so, we believe that we'll be able to complete the project, but it will be slower than we originally anticipated.

One of the requirements set by our client is that we must use InfluxDB, which is a time series database used for time-based data. Since timing is vital in environmental data collection, timestamps will be important for the HyperRail. Because InfluxDB is primarily used for time-based data, it makes non-environmental data storage more difficult as the data stored in InfluxDB must have a timestamp associated with it. Therefore, more external files must be used to store non-environmental data which although is still relatively easy to access through the node server, it may be difficult to manage as the number of files accumulate.

The biggest challenge we've encountered is that Arduino is single-threaded, which will make scheduling data collection runs more difficult. Our initial plan was to have a scheduler thread on the robot that would trigger a run after a specified amount of time, but this won't work with a single thread as it will block the robot's execution. Our next idea would be to have the server avoid sending the run to the HyperRail bot until it's ready for the run to be triggered. This comes with its own set of problems that we'll have to discuss later down the road, but for now it's a stretch goal.