Web-Based Controllers for the ROVOR Observatory

Caleb Gaunt

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Joseph Moody, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2020

ABSTRACT

Web-Based Controllers for the ROVOR Observatory

Caleb Gaunt
Department of Physics and Astronomy, BYU
Bachelor of Science

ROVOR is Brigham Young University's remote observatory in Delta, UT that currently runs on TheSkyX software and other scripts. The current system, though, can be complicated and can only be accessed by one student or faculty member at a time. To solve these problems, a new observatory control system is being developed. The new system, Remote Observe, uses a web-based approach to provide easier access and greater extensibility to the system. Remote Observe has a user interface built using React and is hosted by Google Firebase with a database that tracks commands given to the system. Bret Little has programmed these parts to accept modules designed to interface with different pieces of equipment that utilize ASCOM Alpaca drivers, which present the equipment as an HTTP API. In order to complete a proof of concept, I have programmed a controller for a weather monitor and our custom Lifferth dome using Node.js. I researched the tools and libraries needed to accomplish the development of these controllers. After that, I engaged in a test-first development approach by writing code for automated unit tests that would verify the functionality of the controllers. I then developed the code for controllers until they functioned properly. To validate the system as it is, I found an ASCOM-made Alpaca equipment simulator and connected it the controllers, which I registered with the rest of the existing system. After finding a few bugs and writing a few more tests, I modified the code to where the system integrated with the simulator as expected. In the future, the system will be further developed by obtaining Alpaca-compatible equipment to connect the system to, developing the front-end, rewriting the script that controls the Lifferth dome in Node.js, and configuring the whole thing to run on Raspberry Pis onsite in Delta.

Keywords: observatory, software, internet, programming

ACKNOWLEDGMENTS

Foremost, I would like to acknowledge Dr. Joseph Moody and Bret Little. As my research mentor and physics professor, Dr. Moody was instrumental in my undergraduate education. I greatly appreciate his patience and guidance in helping me find and work through a research project that matched my personal path through college.

Bret Little, a BYU alumnus, software developer, and mastermind of Remote Observe, was invaluable to my involvement to this project. I could not have accomplished what I did without his direction, training, and help. I am especially grateful for the sacrifices he has made to initiate the development of this system and meet with me so that I would have the knowledge I needed to contribute to it.

I would also like to acknowledge Brigham Young University and its College of Physical and Mathematical Sciences, especially the Department of Physics and Astronomy and the Department of Computer Science. Without the resources and opportunities they provided to me, my research contained in this thesis would not have been possible. I would also like to thank the other members of the ROVOR research group, both past and present.

Lastly, I would like to acknowledge the many mentors who helped develop my abilities as a developer as well as the countless contributors to the web development community who created an environment where I never had to learn new software skills alone.

Contents

Chapter 1

Introduction

1.1 ROVOR Observatory

Brigham Young University's Remote Observatory for Variable Object Research (ROVOR) is a simple station set up in Delta, UT to observe for several astronomical research projects (Rovor.byu.edu Accessed Aug 8, 2019c). This relatively small and cheap observatory is controlled by students and faculty remotely by using software. It employs a 16-inch RC Optical telescope and the custom Lifferth dome. This special "dome" is opened by a hydraulic system lifting the roof off the telescope's housing for a greater range of motion and observation with no horizon (Rovor.byu.edu Accessed Aug 8, 2019a).

Brigham Young University has a whole research group, led by Dr. Joseph Moody, that is devoted to using and maintaining ROVOR for undergraduate and graduate research projects. These projects include observation of blazars and other active galactic nuclei, satellite monitoring, and weather forecasting (Rovor.byu.edu Accessed Aug 8, 2019a). Some notable astronomical objects that have been observed for research projects by ROVOR include Mrk 501 and Mrk 421 (Rovor.byu.edu Accessed Aug 8, 2019b), as well as M33 (Byu.edu Accessed Aug 20, 2019).

1.2 Current Difficulties

While the current control system for ROVOR fulfills the research needs for the department's faculty and students, the system also has many difficulties. One of the primary programs ROVOR utilizes is TheSkyX, which has much advanced functionality which goes unused. TheSkyX is in the end more complicated and difficult to use than is desirable, leaving new researchers lost as to what to do or where to start.

The difficulty is further increased by the way that TheSkyX manages their documentation. During the researching phase of my project, I tried to learn how to use TheSkyX by searching for documentation or tutorials. However, I couldn't find anything outside of TheSkyX's website. There they had plenty of documentation, but they restricted access to it unless I already had a product key or paid to view it (Bisque.com Accessed Aug 19, 2019). Especially for students, this can be burdensome and without documentation to read, it can be difficult to learn how to use the system without a mentor taking a significant time to teach them.

Another issue that faculty and students encounter when using the ROVOR system is that their system runs on a single computer. In order to access the observatory controls, a researcher must use a remote desktop program to control the single computer. However, if another researcher needs to access the system from another location to process data, examine logs, or make plans for future observation, they must negotiate with whoever is currently using the computer and take turns. This lack of support for concurrent use can especially cause issues when multiple faculty members and/or students have urgent needs that require access to the system.

1.3 Changes to the System

The purpose of the Remote Observe project is to develop a new web-based control system that would help address these issues and perhaps provide some additional benefits. The first benefit is that the new system has been designed to be simpler and easier to use. To reduce the procedural complexity that TheSkyX brings with it, a new in-house web-based control system would introduce the opportunity to create a user interface that is more intuitive with only the features that are relevant to using the ROVOR observatory. Over time these features may change and more may need to be added. All this implies that the system would benefit from a modular architecture where functional logic could easily be added, removed, or modified withing Remote Observe without risking major changes for the entire system.

In contrast to the limited access to documentation for using the current system, the people developing the new system will be able to write their own documentation for faculty and students to read. This could still lead to a lack of documentation if those developing Remote Observe don't document how it works, but the things they do document would at least be widely available for new users. This along with having an extensible modular architecture, opens the ability for researchers to more directly collaborate on the features they wish to use. By developing a new system for the web, it allows for the integration and use of new technologies that are more widely used in software development today.

A web-based system could also provide support for concurrent users. Instead of using a remote desktop program one at a time, practically any number of students and faculty would be able to log on to an authenticated web page from any location, whether it be on campus, at a conference, or at home. In order to support this kind of use, though, the web-based system would need to have some form of user input management so that the controllers know how to handle inputs that come from many users in quick succession.

1.4 Architecture

The architecture for Remote Observe was designed and largely implemented by Brigham Young University alumnus Bret Little. In this section I will describe the framework he set up for further development (see Fig. 1.1). The code for Remote Observe is contained in several repositories on GitHub. The main repositories for the architecture there are observe-client and observe-web, although other pieces of technology are also included in the design of the control system (Github.com Accessed Aug 2, 2019).

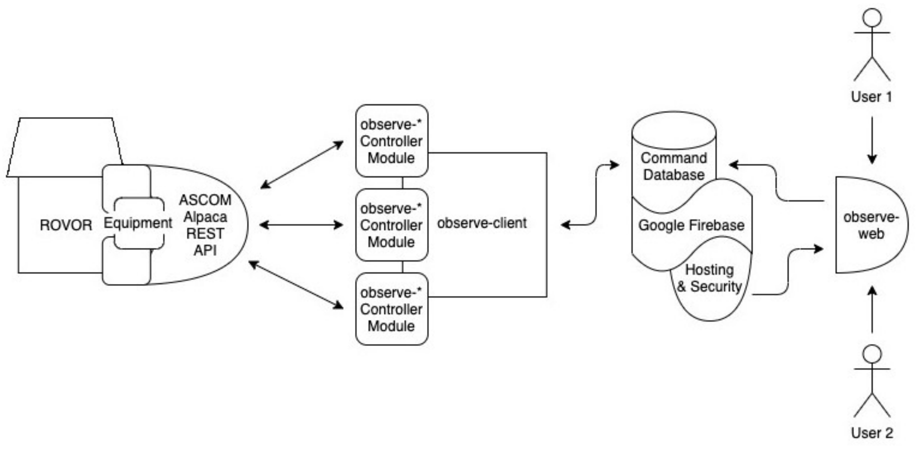

Figure 1.1 This graphic I created visually represents the architecture of Remote Observe, including observatory equipment, back-end design, front-end design, and concurrency of users.

The design starts with observatory equipment and their drivers. The system depends on Astronomy Common Object Model (ASCOM) Alpaca, a technology that was released in early 2019. Alpaca is a standard that equipment drivers implement, which allows them to be presented as a representational state transfer (REST) application programming interface (API) across the internet. In this way, the driver allows each piece of equipment to behave as a web server with endpoints that can be called upon across the web to execute certain commands received in the form of a hypertext transfer protocol (HTTP) request (Ascom-standards.org Accessed Jul 15, 2019). These drivers also have some level of error handling. When the driver receives a request that is malformed or invalid in some way, it will return a response with a 400 status code. When the equipment is in some kind of error state and the driver receives a request, the driver will give back a response with a 200 status and a description of the equipment error in the body of the response. If a request results in an unexpected error within the equipment or the driver, a response with a 500 status will be returned (Ascom-standards.org Accessed Jul 18, 2019).

With the equipment set up to receive HTTP calls, the next part of the architecture is a set of services that a user can use to call the observatory equipment's REST API. To accomplish this, there is a front-end interface that the user interacts with and a back-end that interprets user input and makes the appropriate HTTP call to execute the desired commands. The decision was made to program the back-end in Node.js since it can be run in many kinds of environments, works well as a lightweight scripting language, and has good web capabilities and resources. This way, the back-end can effectively run on an inexpensive computing machine, such as a Raspberry Pi. Most of the back-end code is stored in observe-client, which contains the logic for communication with the front-end.

In addition, observe-client also establishes an interface for controller modules. Each module is responsible for the specific logic for calling one piece of equipment. For instance, there may be a module for calling the telescope's REST API, while another one may be used to control the observatory's rotator through its REST API. These modules are written separately and registered with observe-client using a configuration file. This file tells observe-client which controller modules are available, where to find them in the system, and what data they need to operate correctly.

Each controller module is stored in a separate code repository and contains two functions. The first is getStatus, which is required for implementation. This function sends a number of GET requests to the equipment's REST API to receive information on the equipment's current data and activity and packages it up for the front-end to interpret. The second is the optional executeCommand, which allows the controller to take a command from the front-end. This function then transforms it into an HTTP call to the Alpaca driver, which then performs a corresponding action, such as having a camera start an exposure.

These modules ultimately allow for the system to have the desired level of extensibility. If a new piece of equipment is purchased for ROVOR, all a researcher needs to do to integrate the equipment with the back-end is to write a Node.js module that exports a getStatus function and, depending on the equipment, an executeCommand function. Then once the researcher adds the registration information to the configuration file within observe-client, the equipment will automatically be integrated into the back-end system next time observe-client runs under that configuration.

The front-end code is contained in the observe-web repository, which uses the React programming language to construct a graphical web page (Firebaseapp.com Accessed Aug 2, 2019). This is where relevant equipment data is displayed to the user as well as input forms for the user to use to send commands to the equipment. For such a web page to function well, it needs a platform to host it in a secure manner. The decision was made to use Google Firebase for these features. Firebase runs the React web page and enables the web page to only be accessed through a whitelist of Google account users. This way, users who are not authorized by a Remote Observe admin may not access the web page and are thus unable to control the observatory.

Firebase also provides a database for storing commands from the user and communicating with the back-end. Once a user submits a form on the web page for the equipment to perform a specific action, the observe-web stores the data as a command object with the “sent” status in the database. The running instance of observe-client continually pulls in commands from the database and when it finds one with the “sent” status, it changes the status to “pending” and calls executeCommand on the appropriate controller module with the command’s data. After the execution, the command’s status is changed to “complete” or “error” depending on if an error occurred in the command execution. An added benefit of this database is that if there is a failure somewhere in the system, the database will continue to retain commands coming in from the users, which may then be executed after the system is fixed and running correctly.

1.5 Previous Work

Before my contributions, Bret Little had been the developer for Remote Observe. His work included designing the architecture as well as setting up the boilerplate functionality of the system. On the back-end, he fully implemented observe-client with its command interface, controller registration, and connection to the Firebase database. In Remote Observe's GitHub repositories, he also created observe-weather, a simple controller for a weather sensor. At the time, observe-weather implemented the getStatus function only, but all it did was return a preset JSON object from a file.

Bret also created a working front-end. He set up the Firebase database and hosting, as well as the authentication whitelist, which included himself, Dr. Moody, and me once I started working on Remote Observe. Lastly, Bret developed a working version of observe-web. Because a front-end user interface (UI) can sometimes be prone to breaking after making small changes, Bret decided that the UI should be developed only after the back-end was mostly functional and ready for regular use. In the meantime, he built a web page with no capability for user input. Instead, it simply displayed a JavaScript Object Notation (JSON) object for the database commands and statuses for the registered controllers. This way, the front-end could still display information from the back-end for testing purposes.

Chapter 2

Development

2.1 Plan for Proof of Concept

The task of implementing the full architecture of Remote Observe for all ROVOR's needs is a big one, too large for any one developer to do all at once. As such, my role of the project was to develop Remote Observe enough to produce a proof of concept by implementing the rest of the end-to-end baseline functionality. With Firebase and observe-client already implemented by Bret, I could then focus on developing some controller modules.

It was decided that I would first work on observe-weather, the weather controller module, since it already had a fake implementation with the JSON file and because it would only implement getStatus and not executeCommand. As perhaps one of the simplest controllers, developing observe-weather would enable me to better work out the kinks in the system without having to deal with complex logic that other controllers would potentially require. After I implemented observe-weather, I decided to develop a simplified dome controller, observe-dome, to prove the development process and functionality of an executeCommand function. Because ROVOR uses a custom Lifferth dome, the only necessary command executions to implement were for opening and closing the dome as well as aborting the current slew. With both observe-weather and observe-dome, I would be able to show the capabilities and performance of the system. As an added benefit, future developers would be able to use these controllers as templates for developing new controllers and thus extend Remote Observe's functionality.

Once the two controllers were developed, I decided that to produce a full end-to-end proof of concept, I needed to connect the controllers to observe-client and to an actual Alpaca interface. In the end, I decided to find a simulator that would mimic equipment with Alpaca drivers installed. This way, I could configure the controllers to point to the simulator and run the whole system. This process would then work for system integration testing as well as a proving the viability of Remote Observe as a whole.

2.2 Development Approach

For my work in developing Remote Observe, I decided to take a test-driven approach. This approach consisted of development cycles where I would first choose a small piece of functionality, such as making a REST call, formatting some data, or handling an error. Then I would write the appropriate unit tests to test the new functionality. This would ensure that the necessary tests were in place for validating the functionality throughout the development process. Additionally, doing this would provide additional definition for the functionality being developed when it came to desired results, technical requirements, and development strategy. One other benefit I personally found to this approach was that it enabled me to learn on a deeper level during my development. I certainly made mistakes in some of my designs and implementations. However, this approach helped me find these issues earlier and correct them in a way that helped me learn how to better work with the code and tools provided to me. Ultimately, if the unit tests were written correctly at this point, they would fail when run because the corresponding functionality being tested wouldn't have been implemented yet. All other preexisting unit tests would pass when run.

Next, I would start developing the new piece of functionality. Occasionally I would pause to run the full suite of unit tests to ensure that the code changes weren't breaking previously implemented features. I would then continue to make changes until all the tests, including the unit tests for the new functionality, passed. All this enabled the tests to drive development. They defined what the functionality should accomplish and served to be the means by which I could know that the functionality was complete.

Once I knew that the piece of functionality was complete, I would use Git to commit my changes to the appropriate Remote Observe GitHub repository, which stored the codebase and managed our version control. In this process, I would create a pull request for my changes and assign it to Bret for review. If Bret would find issues, he would notify me of the changes that needed to happen. I would then continue my test-driven process until the issues he pointed out were fixed. Once Bret would find the code changes satisfactory, he would merge the pull request and its code changes into the codebase.

Sometime during or after the code review process, I would choose a new piece of functionality and perform this test-driven development process again for the new functionality. Eventually this would lead to all functionality being developed and well-tested. At some point later, I decided to use the Alpaca simulator for integration testing since the unit tests only showed how well the different parts of code worked on their own as opposed to how well they worked together. Running the simulator and connecting all the services and controllers together allowed me to find other issues in the code. I could then test and fix these issues using more of my test-driven development process.

2.3 Researching Third-Party Tools

Before I could start the development process, I needed to do some research on third-party tools and libraries for development. I began with the tools Bret had already begun using for Remote Observe. Although I had previously learned how to use React (Reactjs.org Accessed Jun 11, 2019) and Firebase (Google.com Accessed Jun 19, 2019) to a degree, I brushed up on my knowledge of these technologies so I could better understand the Remote Observe system. In addition to these, Remote Observe utilized Oclif for managing command objects within a command line interface (Oclif.io Accessed May 1, 2019), ESLint for scanning the code for quality issues (Eslint.org Accessed May 9, 2019), Yarn for dependency management and building the code (Yarnpkg.com Accessed Aug 1, 2019), and CircleCI for code building and deployment (Circleci.com Accessed April 29, 2019). I studied each of these until I sufficiently understood how to use them in the development I would be performing.

The next tool I researched was ASCOM, particularly Alpaca. ASCOM has a decent amount of documentation and videos for their products on their website, which I read and watched (Ascom-standards.org Accessed Jul 15, 2019). I found the Alpaca API documentation to be especially useful since it outlines the format of the Alpaca driver endpoints, including required parameters and the JSON format that they sent back as a response. In addition, the documentation included a small mock server that gave dummy responses for each endpoint (Ascom-standards.org Accessed Jul 18, 2019). This tool became important for development and testing since it allowed me to learn how to form the proper requests to an Alpaca driver and imitate the responses from Alpaca for unit tests.

Most of the other tools I used were third party Node.js libraries for development and testing. Foremost of these was a library that could send HTTP requests. I researched many of them, considering their ease of use, quality of development, frequency of maintenance, use in the development community, synchronicity, and format of code contract. These last two especially defined which libraries would be most beneficial to the project. Some libraries only made synchronous HTTP calls, which meant that they would have to be done one at a time. Since some controllers, like observe-weather, would be making close to a dozen HTTP calls for each command or status retrieval. This meant that an asynchronous HTTP library would be preferable. This way the controller could make every HTTP call all at once and aggregate the data after each HTTP response returned, leading to a faster execution that wouldn't be blocked if a single HTTP request failed. As far as form of contract went, an asynchronous getStatus function, as observe-client defined it, lent itself to a promise-based contract where getStatus would return an object that would eventually contain the status data. In the end, I decided to use Axios as it seemed to best fulfill the needs for developing the controller modules (Github.com Accessed May 24, 2019).

In addition to Axios, I needed several libraries for unit testing. I mainly looked for three types of testing libraries: a unit testing framework, an assertion library, and a mocking library. A unit testing library would be used to set up the boilerplate code for running the unit tests. With this library in place, all that would be needed would be for me to write setup and teardown functions to be run before and after tests as well as functions for the tests themselves. Deciding on this library was relatively easy. When I started working on Remote Observe, Bret recommended that I use the Mocha unit testing framework. After doing some research online, I found that this would be a good option. Mocha seemed to be extremely popular and appeared to have a lot of community support, especially when it came to its use when combined with other testing libraries (Mochajs.org Accessed Jun 6, 2019).

An assertion library is one that provides a set of assertion functions that check certain conditions of the system that the test defines. If all assertions in a test show that conditions are as expected, the test will pass. Otherwise, the test will fail and return a configurable message that provides information about the system and why it failed the test. Mocha already has built-in assertions, but I found them to be difficult to use, not very readable, and limited in what they could assert. This resulted in the need for another assertion library that was compatible with Mocha. After some online research, I found that Chai was potentially the best library for Remote Observe. Chai had a wide range of configurable assertions that were very easy to read and use (Chaijs.com Accessed May 10, 2019a). I did come across one problem, though. Chai largely used assertions accessed through an object's properties, which ESLint considered to be poor coding practice. I tried configuring ESLint to ignore these issues, but it didn't work. In the end, I found dirty-chai, an assertion library identical to the original Chai library except it was designed to circumvent this issue by replacing every property assertion with an identically named function (Chaijs.com Accessed May 10, 2019b). After switching over to dirty-chai and its function assertions, ESLint's code scans no longer reported any code quality issues related to the use of an assertion library.

Mocking libraries are used to create fake objects or programmatic entities that seem to behave like they're real counterparts except that they can be programmed to respond in a preconfigured way. I needed this type of library for unit tests because each controller needed to be connected to observe-client and point to an Alpaca API on a piece of equipment. However, it would be cumbersome to stand up these resources from the controller's code base, especially considering that the unit tests only served to test the controller's own functionality. A mock library would alleviate the need to stand up these resources by creating a fake version of observe-client and a fake Alpaca HTTP server that could be injected into the controller. I looked at a few different mocking libraries and eventually decided to go with Sinon. Sinon provided the ability to override objects, individual functions, and variables as well as provide useful data about the mocked entities such as how often one of its functions had been called. The configurability of these mocked entities was also quite extensive, providing a simple means to override the return value of a function as well as the ability to run functions I created in the place of the actual functions (Sinonjs.org Accessed May 14, 2019). I did run into one minor issue with Sinon, though. This library allows a user to create a fake HTTP server, which would have worked quite well for testing the controllers since they make HTTP requests to Alpaca drivers installed on equipment. However, a known issue with Sinon was that its fake servers wouldn't function in the Node.js language (Stackoverflow.com Accessed May 14, 2019). I decided to keep using Sinon, though, because its mocking functionality was so extensive, that I could use it to mock Axios instead of an HTTP server with relative ease.

I used a couple of other non-programmatic tools in developing Remote Observe, namely Atom and Windows Subsystem for Linux (WSL). Atom is an extensible integrated development environment (IDE) that facilitates an efficient atmosphere for developing in Node.js, complete with a text editor, syntactical autocomplete tools, and plugins for GitHub and other tools (Atom.io Accessed May 15, 2019). I chose to use Atom because it was designed for development in JavaScript-based languages and because I had a lot of prior experience using it. WSL provided me with a Bash terminal on my Windows machine so I could run observe-client and unit tests more easily on a command line (Github.com Accessed may 15, 2019).

Beyond all these third-party tools that I researched, there were a few I did not end up using. Most of these were weather data APIs such as ClearDarkSky (Cleardarksky.com Accessed May 5, 2019) and Dark Sky API (Darksky.net Accessed May 5, 2019). I had investigated these before I found the Alpaca API documentation with the intent of using them for a proof of concept for observe-weather. However, they all required me to purchase an API key and once I found the Alpaca API documentation, using another API was no longer necessary (Ascom-standards.org Accessed Jul 18, 2019). I also researched some Node.js libraries for Raspberry Pis, but I didn't end up getting to the development phase where they would be needed. I will discuss these Raspberry Pi libraries later in the Future Work section of this paper.

2.4 Unit Testing

In this section I will focus on the unit testing I performed throughout my development process. Once I had the necessary tools in place to start a test-driven development cycle, I began work on observe-weather by setting up a testing environment with Mocha. At this time, observe-weather still returned a preset JSON object from a file, so I started by writing a temporary test that simply asserted that observe-weather's getStatus function returned the same JSON object in that file. Normally this test wouldn't be terribly useful. However, I wrote it so I could ensure that I had set up the testing environment in Mocha correctly. That way I could write future tests with the assurance that if they failed, it was due to a failure in the controller code, not in the testing environment's configuration. Initially this temporary test didn't work and had other issues for several reasons, including tool and library incompatibilities. After making some changes and switching out some libraries, such as Chai for dirty-chai, the test passed without incident.

With the testing environment configured, I began writing the relevant unit tests for transitioning observe-weather from its initial file-based functionality to the needed HTTP-based functionality. I ran into another issue fairly quickly, though. The documentation didn't tell me very much about the form of errors I could get back from the Alpaca drivers. It said generally what kinds of information would be passed back in the error HTTP response, but the format of this data was documented in a vague enough manner that I couldn't confidently write a test for or implement error handling logic. This was especially the case for the Alpaca-specific errors, which would have a 200 status. These errors had their own typing system, for which I couldn't find very clear documentation. Without knowing very much about these error types, I decided to stub out these tests to have no testing logic.

In time, I finished developing observe-weather and its tests, which tested a "happy path" for its getStatus function. When I moved on to developing observe-dome, I started by copying over the code from observe-weather, including its tests. Because the two controller modules would have similar code, I figured I would start developing observe-dome as a copy of observe-weather then modify it using my test-driven development approach to match the dome specific logic.

I refactored the unit tests and code for the observe-dome getStatus function quickly, so I moved on to writing the unit tests for executeCommand. Part of the way through writing these tests, though, I came across a problem. I had unwittingly written and organized the unit tests thus far in a way that was cumbersome and difficult to read for larger test suites. However, I hadn't been aware of it before because the testing suite for observe-weather had been smaller. The problem only became apparent when I had started expanding into the many tests for the executeCommand function in observe-dome. Once I saw how cumbersome and repetitive the tests would become, I decided to stop and restructure the way the tests were written in observe-weather. This decision proved to be more necessary than I had originally realized because simply reordering some tests in separate test suites broke all the tests in observe-weather. This level of fragility meant that I needed to make some other improvements that would significantly increase the robustness of the tests. It took a long time to fix the issue that caused the test fragility, but I ultimately discovered that I had made a false assumption about the call order within the Mocha framework. This assumption resulted in me inadvertently placing test setup logic before the teardown logic from the previous test which would clear the environment. This would leave tests in an unconfigured environment, causing them to fail. It turns out that the only reason the tests had passed before is because the order of the tests happened to preserve the test environment. Eventually, I moved the setup and teardown logic to the correct places and finished the refactoring I had started, leaving the test environment much more robust, extensible, and easy to read.

After this long refactor, I finished writing the unit tests for observe-dome, which was followed by its development. Soon after that, I found the Alpaca simulator, which enabled me to learn more about the format of error responses from an Alpaca driver (Github.com Accessed Jul 13, 2019). This enabled me to write out the previously stubbed out error tests. I soon discovered that some of the logic for validating the fake HTTP calls through Axios was repetitive across the tests, so I performed a small refactor to extract the logic for mimicking an Axios call. This made the test code even cleaner and more stable, allowing for easier extension of the test suites.

At this point, I ran into a few more development issues, which I will discuss in the next section.

After making some adjustments, the baseline for the test suites was complete. I decided then to diversify the test cases more with different inputs and configurations such as using input values with different levels of numerical precision, putting errors in status calls, and using some invalid configurations. These diversified test cases were handled by the code quite well and I found no additional issues. In the end, the tests covered nearly all the code paths with an 86% statement coverage for observe-weather and 85% statement coverage for observe-dome. ESLint also found no code quality issues.

2.5 Controller Development

In this section I will focus on the work I did on the controller code in my development process. The first thing I did for development was set up my development environment. I would write the code in Atom and use WSL to run Yarn and Node Package Manager (NPM) for building and running the code. I initially ran into some issues with setting up my Linux environment on my Windows machine through WSL. The Linux environment had a separate file directory from my Windows system, which made using Atom difficult. As well, I tried running observe-client to help me become familiar with it and ensure that my environment had been set up correctly, but observe-client wouldn't run. It turned out that I had several things configured incorrectly, but Bret Little helped me fix these issues. In the end, I had my Linux environment working and was able to get observe-client to run.

With my development environment functioning, my development then started with observe-weather. I initially stripped out the file reading code and used Axios to make a dummy HTTP call to Google. I would then log the response out so I could confirm that the call worked correctly. The code then returned an initial weather object that only contained property definitions for metrics such as sky temperature, humidity, and cloud cover.

With the calls through Axios functioning correctly, I worked to more fully flesh the controller out by configuring it to call an Alpaca API. The ASCOM Alpaca API Virtual Server that comes with the API documentation greatly helped me in figuring out how to form these requests and interpret the corresponding responses in code. I programmed the controller so it would get the observatory's base uniform resource locator (URL) among other pieces of information from the configuration object provided by observe-client. The controller would then initialize the weather data object that the getStatus function would eventually return except that it contained only empty values. The weather controller would then make several concurrent calls to the observatory's Alpaca API, put the result of each call into the weather data object, and return it as the controller status. I didn't know what kinds of weather metrics our specific sensors ROVOR would be able to detect for, so I included all of the metrics that Alpaca supported. I designed the code to be flexible, though, so if one of the calls for a weather metric went awry, it would simply set that metric in the status object as a "null" value. Also, if a need came up to add or remove different weather metrics from the weather status call, the only change that would be needed would be to add or remove them from the initial weather status object for in the code and update the tests accordingly.

When I started the dome controller, I had to do some additional planning because ROVOR's Lifferth Dome is much different than a standard dome. It doesn't rotate and it uses a script written within the Physics Department as a driver. However, I found that ASCOM supports a more limited command API for roll-off domes that matches the functions of the Lifferth dome. Furthermore, I decided to develop the controller as if the dome were running an Alpaca controller since the dome script could potentially be modified to implement the Alpaca standard.

In the end, observe-dome implemented getStatus in a similar fashion as observe-weather. It receives the base observatory URL and other configuration values from observe-client. Then a status object is initialized with property definitions and concurrent HTTP calls are made using Axios to the dome's Alpaca endpoints. When an HTTP response is returned, the status object is updated. Whereas the status object for observe-weather contains over a dozen properties, the status object contains only a string property representing the dome's status and a boolean representing whether the dome can perform a slew. Because Alpaca returns a numerical code for the dome's status, the controller maps the number to a string. A 0 from the Alpaca driver represents an open dome, 1 a closed dome, 2 an opening dome, 3 a closed dome, and 4 a dome in an error state (Ascom-standards.org Accessed Jul 18, 2019).

There are three commands for the dome controller implemented in its executeCommand function, each of which are sent to the Alpaca driver via an HTTP PUT request. In this request, the request body should be passed in through query parameters in the Alpaca URL. The openShutter command simply opens the dome or does nothing if the dome is already opened. The second is closeShutter, which closes the dome or does nothing if the dome is already closed. Because opening and closing the dome takes time, the dome status doesn't immediately go to an "open" or "closed" state when the command is executed. Rather, it first goes into an "opening" or "closing" state until the mechanical operation of opening or closing the dome is completed. The last command that the executeCommand function accepts is abortSlew, which stops the opening or closing of the dome and throws it into an "error" state. At this point, the dome cannot be opened until the dome is fully closed first.

These initial implementations worked well for valid behavior between observe-client, the controllers, and an Alpaca API. However, when I completed these first initial implementations, I couldn't decipher the ambiguous documentation ASCOM had on the error responses from an Alpaca driver. Soon thereafter, though, I discovered the Alpaca simulator, which helped greatly in changing the code to mimic and examine error responses from an Alpaca driver (Github.com Accessed Jul 13, 2019). At this point, I was ready to program the driver's error handling capabilities. It turns out that the request body on an error response was nothing more than a string. For getStatus calls, I was already allowing the failed GET requests to set their corresponding metric in the status object to a "null" value. However, at this time I also decided to add logic that would throw an exception if every status call failed. I figured that if every status call failed, there was likely something wrong with the equipment or system. A similar exception would be thrown if the PUT request failed in executeCommand failed. It took some fine tuning, but eventually I had both controllers programmed to send back exceptions with significant descriptions on the state of the system and nature of the error when an error occurred.

During the development process, I also came across an issue where some of the tests were receiving errors from unhandled rejected promises when an error response was configured to come back from the Alpaca simulator instead of throwing the expected exception. It turns out that there are a few different ways to handle rejected promises and I had been mixing two of those approaches. To fix this, I simply chose one approach and refactored the code to match this approach. After this fix, I ran the tests again and, as desired, the error from the simulator translated to a properly rejected promise, which was caught and repackaged as an exception with the error information. As a side benefit, this refactor also resulted in the controller code becoming much clearer to read and understand.

2.6 Miscellaneous Fixes and Enhancements

At this point, Remote Observe functioned as desired for both normal functionality and error handling. However, there were a couple of small adjustments I made to augment the functioning code. The first came when I went to the GitHub repositories and found several warnings about several potential security vulnerabilities in the code. It turns out that some of the dependencies within Remote Observe needed to be updated. In response, I changed the dependencies in the Yarn files to use more recent versions of the libraries being pulled into the system. With each changed dependency, I ran a full suite of tests to ensure that the new dependency versions didn't break the system. It took several rounds of dependency changes, but eventually the warnings from GitHub went away.

Another change came at Bret Little's request. He saw that for some weather metrics, the Alpaca simulator returned extremely precise numbers. This resulted in the front-end rendering every time a metric changed a trivial amount. To reduce the amount of page renders and thus make the front-end more performant, Bret recommended refactoring the getStatus functions to return values with a certain numerical precision. That way the front-end would only see a change in the data when a more significant change occurred. I wasn't sure what numerical precision should be used, so I programmed it to be configurable for each controller module through the configuration file within observe-client. If a precision wasn't provided, I would have the controllers default to a precision of six significant figures. I couldn't find a good library to perform the rounding up or down to a specified precision at the time, so I wrote my own algorithm. Along the way, I adjusted the existing test cases to include different scenarios, such as having to round values up, round them down, and appropriately round to six significant figures when no configuration was given.

Chapter 3

Conclusion

3.1 Running with a Simulator

When observe-weather and observe-dome had workable initial implementations, I came to a point where I had to decide where to focus my next development efforts. I could have continued writing more controller modules, but if there were some problem in the controllers I had initially written, I would likely write the same problem into the new controller modules. Then once that problem manifested, I would have to fix it in several controllers instead of just the initial two. This potential for having to perform several lengthy refactors and bug fixes led me to avoid this option.

I concluded that the next step should be to connect the modules into a larger system to ensure that they operated correctly in an end-to-end scenario. This testing of the integration between the controller modules and the rest of the system could also act as the proof of concept this project sought to produce. However, the only way to do this would be to connect it to a running Alpaca API. The apparent solution for this would be to hook up the controllers to a real Alpaca-run weather sensor and dome. This would take quite a bit of time and money, though, since ROVOR didn't have a working weather sensor at that time and its custom-built dome wouldn't have a commercially available Alpaca driver.

There was another possible way to do this end-to-end integration testing, though. If I could find a simulator that imitated an Alpaca API, I could point the controllers at it to create a "full" observatory system for the purpose of proving the viability of Remote Observe. My first thought was to use the virtual server that came with the online Alpaca API. At that time, I configured the system to point at this server and for a short time, the system operated correctly. However, after about a dozen seconds of runtime, the virtual server started sending back errors for every call. It turns out that their virtual server enforced a strict rate limit, which Remote Observe's status checking easily exceeded. Besides that, the virtual server always returned the same dummy responses, so I wouldn't be able to test against changing statuses or errors from the "equipment" (Ascom-standards.org Accessed Jul 18, 2019). In the end, I learned all I could from these tests and looked for a different simulator.

The next place I looked was among the downloads available on ASCOM's website. They had several drivers, and a few links to some GitHub repositories. At first, I didn't find any program with the Alpaca name attached to it. Eventually after searching through several online forums for recommendations on viable simulation tools, I went to look at the code repository for ASCOM Remote, the precursor to Alpaca. Among the several components in the repository, I decided to install the code for ASCOM Remote Client, which appeared to contain several simulators. I came across a dome simulator within the installed repository that opened a widget that enabled me to control a "dome" using buttons on the widget (Github.com Accessed Jul 13, 2019). However, I needed a simulator to receive and respond to commands programmatically, not a widget to give the commands themselves. Nevertheless, I followed the directions in the documentation for connecting the widgets together into a unified interface. Each time I did, errors came up that I couldn't decipher. After several attempts to launch the program, as well as failing to find information in documentation and forums for how to fix the errors, I decided to abandon these simulators.

The ASCOM Remote GitHub repository also had a folder for ASCOM Remote Server, which I then downloaded and installed. After searching through the files for the server, I found some materials enclosed that referenced Alpaca. I found the launcher for the server and started it up with relative ease. Much to my delight, I found that the server could be configured with different pieces of proxy equipment and would serve on a specific URL. When a piece of simulated equipment was added to the server, it would produce an additional window where the state of the "equipment" could be modified (Github.com Accessed Jul 13, 2019). I configured the server to host locally on a specific port and added a "dome" and "weather sensor". Then I modified the configuration file for observe-client to point at my machine's local port where the server was hosting. When I started up observe-client, requests and responses began appearing on the logs for both programs, indicating that they had successfully connected.

After fixing some bugs that using ASCOM Remote Server revealed, Remote Observe was ready for a functionally end-to-end run as a proof of concept. I began by adding commands to the Firebase database for opening the dome, closing the dome, and aborting the slew. I then started up ASCOM Remote Server and observe-client, which began feeding status information to the front-end web page. To mimic user input to execute a command, I would change the status of the corresponding command in the database to "sent". To mimic a change in weather conditions, I would adjust the numbers ASCOM Remote Server's weather window and click the "Enable Changes" button. For the proof of concept run, I put the system through the following scenarios: increasing and decreasing the numbers for weather metrics, opening the dome, closing the dome, aborting the slew while opening the dome, aborting the slew while closing the dome, opening the dome in an aborted error state, closing the dome when in an aborted error state, and disconnecting the "weather sensor" with the system running. In each scenario, the equipment status on the front-end web page updated correctly at the correct times. The logs produced also contained the expected information, including details on the properly handled errors. With all scenarios playing out as desired, Remote Observe had proven itself to be a viable system for observatory control.

3.2 Future Work

There is still much work to do on the Remote Observe project, starting with the controllers. The goal is for Remote Observe to eventually control the whole ROVOR observatory. This means that more controller modules will need to be developed since only the weather and dome controllers have been implemented. These other controller modules would ideally include controllers for ROVOR's telescope, rotator, camera, switch, and perhaps other equipment.

There is also some architectural work that could be done with the project from here as well. I noticed after developing observe-weather and observe-dome that they had a lot of code in common. This got me wondering if there was a good way to reduce code duplication. I considered writing an abstract controller with most of the common logic. This way, new controllers would need even less code to get plugged in and if a change had to happen in the common logic between the controllers, it would only have to be made in one place as opposed to being made for each controller.

However, the more I thought about it, the more I found that it would potentially be more difficult than I thought. The first is a matter of how observe-client and controllers would access it. It could be its own module to be imported, but it would be odd to have such a small module stored in a repository that cannot be run on its own. It could alternatively be included in observe-client, but it would require some potentially intense reworking of the code just to accommodate the abstract controller.

As far as the actual code goes, executeCommand lends itself well to an abstract controller. However, the getStatus code would be very problematic despite being very similar across controllers. This difficulty comes from some controllers having status retrieval calls that require extra parameters that most other controllers don't, such as switch values and telescope axis rates. These extra parameters would have to be included for every controller, even though it would almost always have a null value. Furthermore, each separate call for part of the status would potentially need to be handled and transformed in very different ways. The best way I could think of to handle this would be for the abstract controller to receive several handling functions, which could get very illegible and confusing very fast.

In the end, I decided that the code will be more readable, less confusing, and easier to write if each controller had its own unique logic as I've done, despite there being a high amount of code duplication in this approach. This doesn't mean that an abstract controller shouldn't be written or couldn't be implemented in a clear, readable, and concise manner. Rather, I haven't figured out a way to do it. Perhaps in the future, more consideration and discussion could occur to figure out how to effectively implement such a controller.

As functional as the controllers currently are, they are missing a major feature. The whole system lacks a robust logging system. At this point, logging statements are made out to the console running observe-client, which contains only the configuration objects, requests, responses, and errors that occur in the system. However, these logs are not persistent and would be unavailable if observe-client were to stop running. Thus, a system needs to be put into place that will store and manage these logs. In addition, modifying the current code to have a uniform and clear data logging system would make managing these logs much easier.

As part of this log management effort, Alpaca has functionality for transaction IDs on status retrieval and command execution calls. Alpaca controllers themselves provide unique IDs from the driver on the responses to the controller with the option of receiving separate transaction IDs from the controller as well. This way, HTTP calls within logs can more easily be traced to activity both on the Alpaca controller and the controller. However, Alpaca leaves it to the consumer of their API to determine how they will manage these IDs (Ascom-standards.org Accessed Jul 18, 2019). Considering how and where this management should occur, whether in the controller, observe-client, or a new separate logging module would be another starting place for further development of the Remote Observe project.

Besides further development on the controllers, another place for future work in the Remote Observe project is in making the system more end-to-end. I can think of a few ways of doing this. One apparent way is to connect the system to some real observatory equipment instead of a simulator. Because ROVOR's dome is custom-made for the observatory, I believe that it is less likely to be replaced, which leaves weather equipment for consideration. After purchasing a weather sensor, a more robust system test could be performed by setting up the driver to serve the proper Alpaca REST endpoints on the internet and configuring Remote Observe to point at that API.

As far as purchasing the sensor goes, I did some research and found several Alpaca-compatible weather sensors for consideration in the Remote Observe system. The first was the Boltwood Cloud Sensor II from Diffraction Limited. It seemed like it would go above and beyond for providing for ROVOR's needs, but at a higher cost (Diffractionlimited.com Accessed Jul 18, 2019). The second was an MBox, which was much cheaper but was unable to monitor cloud cover (Astromi.ch Accessed Jul 18, 2019). The third was a Lunatico AAG CloudWatcher, which was moderately cheap and sensed for cloud cover, but not wind speed (Lunaticastronomical.com Accessed Jul 18, 2019). While these appeared to be Alpaca-compatible sensors, there was little enough information on the internet to where I couldn't find out how much support they have for Alpaca. From what I could tell, though, any of these could function with Remote Observe. Which sensor to purchase would depend on how those funding the project would prioritize different weather metrics and price.

Another way to further the end-to-end use of Remote Observe would be to do more development on the React front-end. Currently it only displays JSON data from observe-client and is unable to receive user input. Since the UI of a page can be fragile when back-end changes are made, I would be hesitant to do too much design and development in this area. However, I can think of a few aspects of the front-end that would be appropriate to work on at this point in the Remote Observe's development. To start with, small reusable UI components, such as buttons, forms, and display sections, could be developed or brought in from a third-party library. Once these smaller more modular have been created, it would be feasible to add a simple first attempt at a display section to the web page. I would recommend creating a display section for the dome since it only has two fields to its status and simple controls could be implemented with three buttons in the UI, one for each command. I would also recommend not removing the JSON from the front-end display since it still provides a lot of useful debugging information. I think it would be reasonable, though, to add some logic to the page to toggle hiding the JSON. Then once the front-end page is ready for steady regular use, the JSON could be removed entirely if desired.

There are a few miscellaneous parts of the system that need work for it to work the as desired for ROVOR. The first is to write a new "driver" for the dome. Because the dome was custom made, there wasn't a commercial driver available for it. Currently, it's running with a LabView script that was written in-house (Rovor.byu.edu Accessed Aug 8, 2019a). This presents a unique problem for the Remote Observe system since there is likewise no commercial Alpaca driver for ROVOR's Lifferth dome. To solve this issue, someone will have to write another driver script that presents itself as a REST server that implements the Alpaca standard. After discussing the issue with Bret Little and Dr. Moody, the decision was made to explore a path where a Node.js driver script is run continuously on a Raspberry Pi in the observatory.

I decided to spend some time researching different Node.js libraries that could run on a Raspberry Pi to control the dome through electronic manipulation of the computer's general-purpose input/output (GPIO) pins. Unfortunately, I couldn't find much comparative information on the many GPIO libraries in Node.js, so I went to the documentation of all the libraries I could find. One of the first libraries I examined was wiring-pi, which comes in several languages with a disclaimer about its complexity. This library is quite advanced and most of its functionality was hard for me to understand. Because of this complexity and the simple needs of the driver script, I believe that wiring-pi would be a poor choice for the GPIO library (Wiringpi.com Accessed Aug 14, 2019). The onoff library also seemed a little complicated, but not egregiously so (Npmjs.com Accessed Aug 15, 2019a). What seemed to be one of the simpler libraries for me was rpio. However, rpio seemed to potentially have too much reliance on C++ to be realistically used (Npmjs.com Accessed Aug 15, 2019b). Another library, rpi-gpio, looked promising due to it appearing to be even simpler than rpio (Npmjs.com Accessed Aug 16, 2019b). The last library I considered was pigpio, which had a lot of extra and potentially useful functionality at the expense of being a little less simple (Npmjs.com Accessed Aug 16, 2019a).

I did find one article that compared many of these libraries by their speed. While onoff and rpi-gpio were the simpler libraries, they were also by far the slowest. On the other hand, rpio was at the top when it comes to speed. Despite its drawbacks, this certainly made rpio look like a more attractive option. This article, though, did not list pigpio, so I wasn't not sure how fast it runs (Github.com Accessed Aug 16, 2019).

After further looking over these Node.js GPIO libraries, I narrowed them down to three recommendations. The onoff library seems to be quite popular and although it's slow, it's also very simple to use. The primary benefit that this library has over the others is that it has both synchronous and asynchronous function calls, so if the system ends up needing asynchronous functionality, onoff could serve it (Npmjs.com Accessed Aug 15, 2019a). On the other hand, if synchronous calls work well enough, either pigpio or rpio may work better. If the dome script is sufficiently complex or needs greater extensibility, onoff would probably be the best option. Otherwise the better synchronous option seems to be rpio (Npmjs.com Accessed Aug 16, 2019a). It's the fastest GPIO library in the speed comparison I found and is above average when it comes to simplicity of use (Github.com Accessed Aug 16, 2019). However, if the dependencies for rpio make it incompatible with the rest of the system, I believe pigpio would be the next best synchronous choice (Npmjs.com Accessed Aug 15, 2019b).

Since Remote Observe is meant to have concurrent users, there may be some need for an additional level of command management logic implemented. For instance, if one user attempts to open the dome and another tries to close the dome at the same time, it could result in some unexpected outcomes for one or both users. Thus, more consideration is likely needed to determine how to manage concurrent commands to the observatory. This issue could be helped with some new features, including locking the system when a command is made and providing more information in the UI about who is logged in and what commands they are executing.

Another augmentation that could be researched for this system would be around Raspberry Pi clusters. The observe-client code was meant to be run on a Raspberry Pi, but if it only runs on one, the entire system would be offline until someone went to its location and replaced it. Remote Observe would only go back online after configuring the new Raspberry Pi, hooking it up, and setting it to run observe-client. Thus, it would be useful to have a cluster of several linked Raspberry Pis for redundancy. This would make the system's physical maintenance easier and reduce the amount of system downtime when something needs to be modified or fixed.

Bibliography

Ascom-standards.org. Accessed Jul 15, 2019, https://ascom-standards.org/Developer/Alpaca.htm

—. Accessed Jul 18, 2019, https://ascom-standards.org/api/

Astromi.ch. Accessed Jul 18, 2019, https://www.astromi.ch/product/mbox/

Atom.io. Accessed May 15, 2019, https://atom.io/

Bisque.com. Accessed Aug 19, 2019, https://www.bisque.com/downloads/categories/documentation/

Byu.edu. Accessed Aug 20, 2019, https://rovor.byu.edu/news/

Chaijs.com. Accessed May 10, 2019a, https://www.chaijs.com/

—. Accessed May 10, 2019b, https://www.chaijs.com/plugins/dirty-chai/

Circleci.com. Accessed April 29, 2019, https://circleci.com/docs/

Cleardarksky.com. Accessed May 5, 2019, https://www.cleardarksky.com/

Darksky.net. Accessed May 5, 2019, https://darksky.net/dev

Diffractionlimited.com. Accessed Jul 18, 2019, https://diffractionlimited.com/product/boltwood-cloud-sensor-ii/

Eslint.org. Accessed May 9, 2019, https://eslint.org/docs/user-guide/getting-started

Firebaseapp.com. Accessed Aug 2, 2019, https://remote-observe-dev.firebaseapp.com/

Github.com. Accessed Aug 16, 2019, https://gist.github.com/jperkin/e1f0ce996c83ccf2bca9

—. Accessed Aug 2, 2019, https://github.com/remote-observe

—. Accessed Jul 13, 2019, https://github.com/ASCOMInitiative/ASCOMRemote

—. Accessed may 15, 2019, https://github.com/mskyaxl/wsl-terminal/

—. Accessed May 24, 2019, https://github.com/axios/axios

Google.com. Accessed Jun 19, 2019, https://firebase.google.com/docs

Lunaticastronomical.com. Accessed Jul 18, 2019, https://www.lunaticastronomical.com/p7007858-aag-cloudwatcher-cloud-detector.html

Mochajs.org. Accessed Jun 6, 2019, https://mochajs.org/

Npmjs.com. Accessed Aug 15, 2019a, https://www.npmjs.com/package/onoff

—. Accessed Aug 15, 2019b, https://www.npmjs.com/package/rpio

—. Accessed Aug 16, 2019a, https://www.npmjs.com/package/pigpio

—. Accessed Aug 16, 2019b, https://www.npmjs.com/package/rpi-gpio

Oclif.io. Accessed May 1, 2019, https://oclif.io/docs/introduction

Reactjs.org. Accessed Jun 11, 2019, https://reactjs.org/docs/getting-started.html

Rovor.byu.edu. Accessed Aug 8, 2019a, https://rovor.byu.edu/about/

—. Accessed Aug 8, 2019b, https://rovor.byu.edu/publications/

—. Accessed Aug 8, 2019c, https://rovor.byu.edu/

Sinonjs.org. Accessed May 14, 2019, https://sinonjs.org/

Stackoverflow.com. Accessed May 14, 2019, https://stackoverflow.com/questions/47238726/sinon-fake-server-not-intercepting-requests

Wiringpi.com. Accessed Aug 14, 2019, http://wiringpi.com/

Yarnpkg.com. Accessed Aug 1, 2019, https://yarnpkg.com/getting-started