Developing a Single Page Web Application to Monitor and Control a Solar Powered Autonomous Boat

Morgan William Arthur Trench
21122107
Master of Professional Engineering (Software Engineering) 2019

Word Count
5309

Supervisor
Dr Thomas Bräunl

Research Department
Robotics & Automation Lab
School of Electrical, Electronic and Computer Engineering, University of Western Australia

Submitted 2019

Contents

Abstract

This document describes work completed for the Solar Powered Autonomous Boat/Raft project. The existing web interface for controlling the boat over HTTP wasn't able to be easily modified to handle changes being made to the boat itself. To address this complication, development of a Single Page Application (SPA) was proposed. Developed using the Angular platform, the basic design of the application is outlined in this document. The application program interface (API) that supports communication with the boat was modified to also communicate with instances of the SPA, as well as support extra sensor data originating from the boat. The intended usage of each of the endpoints of the API is also described. A functional control application was created, however real world testing was not achievable due to confounding factors.

Introduction

Background

Autonomous and electric vehicles are gaining in popularity in recent years, motivating factors include environmental concerns, rising fuel costs and their growing availability in the market. Tesla now offers five models of completely electric cars [20], with competitors bringing a number of competitors to market [21]. Fully autonomous 'autopilot' driving for Tesla vehicles has been announced to be ready within a year [22]. The automotive industry isn't the only industry set to benefit however.

Flying delivery drones are on the near horizon from large shipping companies such as Amazon and UPS, with the potential to revolutionize the shipping industry, driving down costs through by eliminating the "last mile" costs associated with delivery [23]. Amazon Prime Air aims to be able to deliver any suitable product under 5lbs (~2.3 kgs) in under 30 minutes using drones. Wing, an initiative from Alphabet, has started delivering goods in Canberra's north, this year [24]. Domino's Pizza as well is in on the action and as a proof of concept delivered two pizzas via drone in New Zealand two years ago [2]. Previously Dominos announced a land based autonomous delivery vehicle, the Domino's Robotic Unit or 'DRU' in March of 2016.

Autonomous vehicles can also be used for surveillance purposes by the police or military. Simultaneously reducing cost of missions and improving response times all without endangering lives of operatives in the field [25]. Searching for mines on both land and sea is a prime example of utilising autonomous vehicles or drones for safety purposes. The use of autonomous vehicles promises benefits for the scientific community as well. The ability for take measurements or retrieve data from installations over large or unreachable areas presents opportunity for better modeling of meteorological or ecological phenomenon [26]. Autonomous vehicles used for these purposes can function individually or as part of a distributed network [27].

The interest of this project is the use of unmanned vessels to perform tasks such as bathymetric mapping or measurements at sea of temperature, salinity and other quantities. For measurements at sea, typical methods include the use of “satellites, airplanes, buoys, research vessels or ships of opportunity“ [28 p2]. These methods suffer from inflexibility (fixed position or route), interference (satellites and planes) or high cost. A relatively cheap alternative would be well received.

The Microtransat Challenge

A competition whose purpose is to stimulate the development of autonomous boats of varying classes and divisions. The aim is to have autonomous boats race from Europe and the Caribbean or North America and Ireland. Participants race in a time trial manner (asynchronously). Exact start and finish locations are posted on their website as GPS coordinates, as well an up to date set of rules [29]. The SB MET became the first competitor to successfully complete the race on the 28th of August after a 79 day journey. Prior to this success, there were 23 unsuccessful attempts from competitors dating back to the start of the competition in 2010. The SB MET and all previous competitors were classed a “sailing vessels”. Since then there have been eight further registrations to the competition, only two of which are classed and “non-sailing”, “That'll Do Two” from Epsom College and "Peruagus” [30].

Sea Charger

Alternate to the sailboat designs common in the Microtransat Challenge, Sea Charger is a project inspired by it that traversed a different route. Constructed from fibreglass, Sea Charger utilises a propellor for ts propulsion, rudder for steering and two large solar panels as it’s power source [31]. Communication with Sea Charger was achieved using a two way satellite modem.

Figure 1: Photos of Sea Charger [31]

Sea Charger succeeded in a journey from California, USA to Hawaii on July 22nd 2016, taking 39 days to do so. It was then re-launched with a new goal of reaching New Zealand. The rudder failed around 500 kms from New Zealand on November 18 2016, after a grand total of five months in the water [32].

Project Outline

The Solar Powered Autonomous Boat/Raft is an ongoing project in the robotics lab at the School of Electrical, Electronic and Computer Engineering at UWA. The ultimate goal of which is to create a vessel that can traverse an ocean and collect sensor data, all the while navigating autonomously to waypoints set by those who deployed it. Under the supervision of Thomas Bräunl, the boat has been contributed to by three students prior the start of this project, John Hodge, John Borella and

Aaron Goldsworthy[3], and presently is being worked on by Morgan Trench and Wesley Coleman. An intermediate goal of the project is to have the boat make a return trip from Perth to Rottnest island.

Previously completed work

John Hodge and John Borella together produced a working initial design for the system, including the hull structure, power and propulsion systems as well as integration of the ArduPilot for waypoint navigation. The boat was proven functional on multiple occasions, first in the UWA pool where any minor issues were ironed out (such as radio frequencies and motor polarity) and then later in the Swan River [1] [2]. Aaron Goldsworthy worked on adding a Raspberry Pi Nano and 3G modem onto the boat, to allow the platform to connect with a server over HTTP to send/receive data and waypoint information, which was successfully demonstrated in his thesis [3].

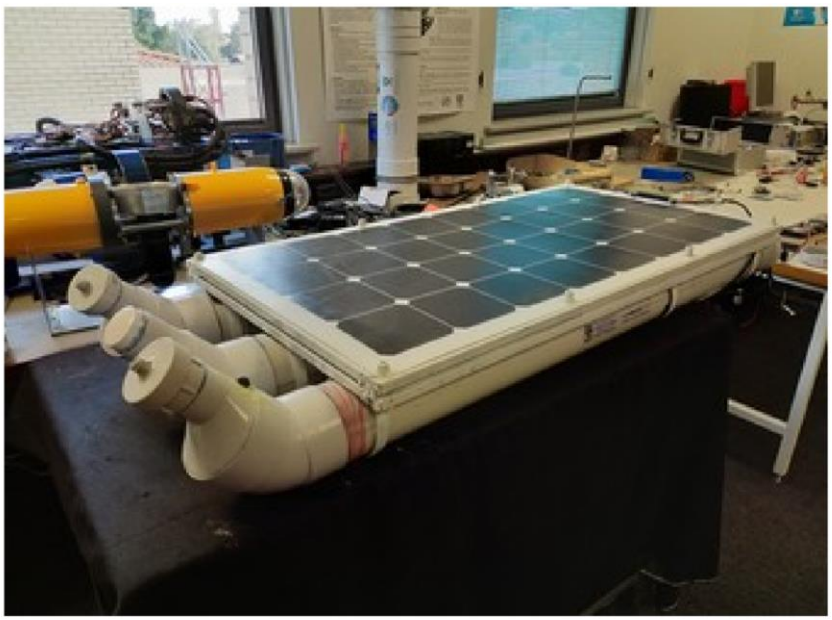


Figure 2: Solar Powered Autonomous Boat at UWA [3]

Starting State

Upon joining the project the second semester of 2018, Aaron Goldsworthy was halfway through his is work establishing communications between the boat and server. He created multiple dynamic web pages, constructed by CGI scripts upon request that could be used to view boat position data and update the commands/waypoints that the boat would navigate between. At the same time Wesley Coleman joined the project with the intention of adding a number of sensors to the platform for quantities such as water temperature, water salinity, atmospheric pressure, power indicators and a camera.

Intended Contribution

With the evolution of the platform to include more sensor data, modifying the CGI scripts that generate the web pages responsible for monitoring and controlling the boat would become much more difficult. Instead the primary aim for the project was to replace the CGI scripts generating web pages with a Single Page Application, servable as a collection of static files. Envisioned features of the SPA included:

- A 'Live' representation of the boat's path on a map
- The ability to set and manipulate waypoints/tasks for the SPAB to follow for authenticated users
- Viewable and exportable sensor data (such as photos, temperature, salinity)
- Energy usage/generation statistics
- Incorporation of freely available AIS data on the map, to help with route planning

To achieve this, changes to the existing API that facilitates communications with the boat will need to be made, and likely change to the the boat software itself.

Literature Review

RESTful APIs

REST an acronym for Representational State Transfer, it is a architectural style (not a standard) used when creating Web Services. It can make use of other internet standards, such as URL [6], HTTP [7], XML [8] or JSON [9]. It is defined by six guiding principles [5] [1]:

1. **Client–server**: Separate the concerns of the user interface from those of data storage. This means that either can be modified or swapped out with minimal change to the other.
2. **Stateless**: Every request from client to server must contain all of the necessary information for the request to be processed correctly, that is the server must be stateless. Any state is therefore on the client's side.
3. **Cacheable**: All responses must be labeled so that the client knows if it is cacheable. If it is it means that subsequent identical/equivalent requests can be forgone and the cached response used in its place.
4. **Uniform interface**: encapsulates four sub-constraints that dictate some specific that help to allow the parts of system architecture to evolve separately
    - 4.1. Identification of resources in requests
    - 4.2. Manipulation of resources through representations
    - 4.3. Self descriptive messages, that is information on how to process the message is included in the message or its metadata

    4.4. Hypermedia as the engine of application state, which means that a rest client should be able to dynamically discover server provided links which provide information about currently available actions.

5. **Layered system**: The client should be unable to tell if it is communicating with the 'end server' or an intermediary. Such as one 'duplicate' servers created to enable scalability
6. **Code on demand (optional)**: Servers can provide code to clients (Compiled Java applets or executable JavaScript) to extend their functionality.

RESTful APIs typically use HTTP as the mechanism by requests are made. What follows is a brief outline the common HTTP request method types used when constructing a RESTful API.

- GET requests are used to retrieve a representation of a resource. A GET request should not alter the state of the resource, subsequent requests should return the same representation of the resource unless it has since been altered with another type of request. The response code should be "200 (OK)" if the resource is found, else "404 (NOT FOUND)"
- POST requests are used to create a resource. Repeated requests should result in multiple created resources. If the resource created is now accessible via a URL then the server response should be "201 (CREATED)" with a reference to the object. Otherwise the response can simply be a "200 (OK)"
- PUT requests are used to update/replace a resource, though it a resource doesn't already exist it can be used for creation or ignored. Typical response codes include "200 (OK)", "204 (No Content)" or "201 (Created)"
- PATCH requests are used to partially update or modify an existing resource, similar to PUT.
- DELETE requests delete a resource at a URI, subsequent requests should fail. Successes should respond with "200 (OK)", "404 (NOT FOUND" if there is no resource to delete in the first place.

Single Page Applications

Traditional web applications refresh the page every time it communicates with the web server that served it. In contrast, Single Page Applications (SPAs) are web pages that are loaded once, and then make subsequent AJAX (Asynchronous Javascript and XML) requests whose responses contain only data to allow the page to modify itself rather than trigger a page refresh. Figure 3 outlines the difference between and SPA and a traditional website/page.

This type of design is quite compatible with a RESTful backend api, as the SPA itself can be served as static files to clients, which make subsequent requests to the RESTful api to get data as needed.

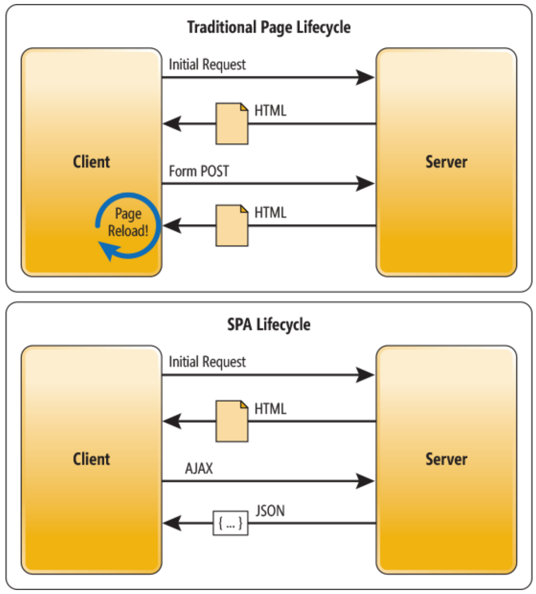


Figure 3: Traditional Page Lifecycle vs SPA Lifecycle [4]

Frontend Frameworks

Frontend web frameworks exist to speed up web development. They solve a number of regularly encountered problems when developing web sites and are particularly useful when making SPAs. Some common benefits include:

- Components - Many frameworks allow you to split up your code into components that deal with specific purposes or functionality
- State Management - Many frameworks (or framework plugins) provide ways of storing application state in a way that is consistent across the application.
- User Interface - Frameworks often have UI libraries associated with them, that help with quickly achieving a clean, responsive interface. Components allow for 'base styles' to be applied and extended, helping both new and experienced designers.
- Reliability - Frameworks are used by many and as such are tested solutions to common problems, and are frequently updated.
- Cross browser compatibility - A bane of web development is supporting older browsers, frameworks often automatically have fallbacks for when desired features aren't supported.

Figure 4 shows a snapshot of the 'Frontend Framework Landscape' over the course of this project. Web frameworks as a category was only introduced in the 2019 survey so comparison is a little difficult, via combination of the two surveys it can be seen that there currently exist three 'major' frontend frameworks, Angular, React.js and Vue.js. Angular is currently used the most by professionals, but it has been losing ground recently to Vue.js and React.js. React.js is an open source framework backed by Facebook, considered a highly un-opinionated in comparison with

Angular and Vue.js. Part React's growth can be attributed to the popularity of '*React Native*', a project for native app development that uses React.js at its core [35]. Vue.js currently occupies the lowest position amongst professional developers but is gaining ground quickly, it is currently the smallest in size compared to the other frameworks [36].

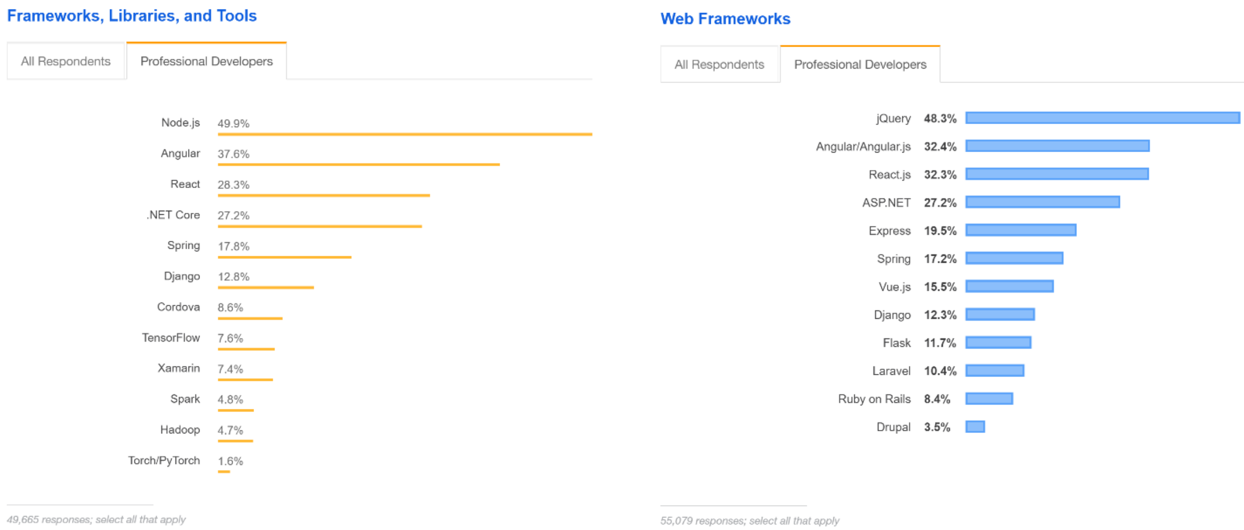


Figure 4: Images taken from the Stack Overflow Developer Survey
(Left) 2018 survey [10], responses collected from January 8 - February 28
(Right) 2019 survey [11], responses collected from January 24 - February 14

Angular

Angular is a platform/framework written in TypeScript for building client-side web applications. It is the successor AngularJS, another frontend framework developed by Google. Angular markets itself with "declarative templates, dependency injection, end to end tooling, and integrated best practices" [12]. Angular is used in development not just for the web applications by also mobile applications though tools such as Nativescript [13], or for the desktop using Electron [14].

Angular applications are comprised from a set of NgModules, that provide a context for compilation for related parts of the application. NgModules combine services and components to form functional units. NgModules are different from JavaScript modules (ES2015), but they are similar in that NgModules can import functionality from one another. An example of a commonly used NgModule is *Angular Route*, which allows SPAs to retain the familiar URL address navigation behavior and history that people are used to [15].

Components are parts that can be declared as part of an NgModule. They define views, which are collections screen elements that can be modified to display application data and respond to user input. Components are defined using HTML templates, CSS (or equivalent) and a TypeScript class that defines the logic behind it. The HTML template can contain "binding markup" to capture events on HTML/DOM elements (event binding) or their values (property binding). Components can depend on Services for data and functionality.

Services are also elements declared as part of an NgModule, they provide specific functionality to Components that may be shared. Services are provided to Components through Dependency Injection (DI), which helps to promote code modularity, reuse, and efficiency. Services are implemented as TypeScript classes. The classes that define both Components and Services are marked with Decorators (metadata), that tell Angular how to use them with information such as their type, and which HTML and CSS files to associate with them.

Angular promotes a Command Line Interface (CLI) tool that is available for use alongside it. Available from the node package manager (npm), the Angular CLI allows developers to quickly scaffold out their applications [33]. Usable with the command *"ng"*, a basic 'hello world' level application can be stubbed out simply by executing the command *"ng new <app-name>"*. New Components and Services (and other elements) can be stubbed out and imported properly with "*ng generate <schematic> [options]*" where the schematic can be component, services, directive or (much) more. Angular Material is a set of premade components for Angular applications to fulfill common UI needs [34].

Common Gateway Interface

The Common Gateway Interface (CGI) is a way for HTTP servers to run external programs in a platform independent way. It has been used in the web since 1993 and can be used to generate dynamic content for responses to HTTP requests [16]. CGI programs/scripts are written to handle requests, after receiving a request the web server starts up a process to run CGI program, or an interpreter to execute the script. The kind of interpreter used in the case of scripts is determined by looking for the Shebang character sequence ("#!") at the beginning of the file that is followed by the path the executable interpreter. The interpreter then runs the script and the server uses the output as the response to the HTTP request. This is illustrated in Figure 5. CGI scripts are usually placed in a dedicated directory, such as "cgi-bin", but they can also be placed deliberately throughout the file system and executed in response to requests to that path/URL.
CGI applications/scripts are usually written in C/C++, Java, Perl or Python.

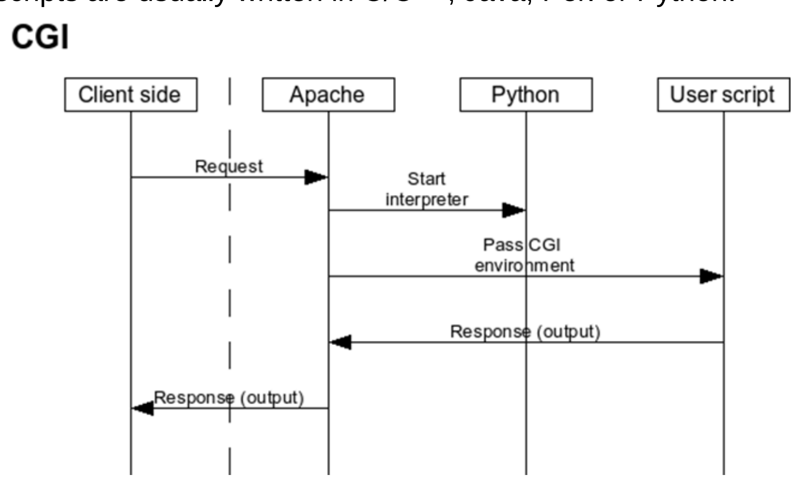


Figure 5: CGI Script execution (Python) [17]

CGI Scripts do have some notable weaknesses, one being that they spawn a separate process for each request. This make them weak against denial of service attacks or spikes in activity.
Fast-CGI is a variation on CGI that has the interpreter/application startup once and handle all requests as they arrive. Alternative to both of these, running dedicated server software specifically for handling dynamic content, such as Express/Node.js or Flask/Python alongside a tradition HTTP server is possible. Often achieved by assigning the 'dynamic server' to run on a separate port and having the traditional HTTP server communicate over that port with it over a protocol such as wsgi or via reverse proxy to localhost [18, 19].

Integrated Changes

Backend

The existing solution for communication with the boat had the boat issue HTTP requests to the server to both send position data and retrieve a list of waypoints to traverse. This functionality remains the same, but the form of the data passed around has been changed. The major changes to the server are to do with removing the existing interface replacing it with an API that the SPA can make calls to. This API exists in a subdirectory of the 'solarboat' project directory that is served by the web server. What follows is an enumeration of the various API endpoints, and their intended functionality. Each endpoint has different functionality, depending on the HTTP method used with it. Example requests and responses are given in appendix figures 15 and 16

/api/commands.cgi

The command endpoint exists to manage to instructions or 'commands' that the boat will follow on a mission. For reference, a valid command for the boat is considered to be a JSON object with the following fields: action (string), latitude (float), longitude (float) and duration (integer).
Valid actions currently only include "moveTo", but the idea is to expand the boats capabilities to handle actions such as "waitAtFor", "driftFor", etc.

PUT

This endpoint is used by the frontend to update the commands. The body of the request should be a JSON array of commands, where each value is a valid json object with fields corresponding to a valid command. The contents of the request, provided each command is valid, replaces the existing list of commands on the server. Upon storage, each command is assigned a random unique identifier (UUID [37]).

GET

Used by both the SPA and boat, this endpoint retrieves the list of commands on the server, that is, commands yet to be completed. The response is a JSON object, containing a JSON array of commands with the uuid added as an attribute.

DELETE

This endpoint is used by the boat upon completion of a command/waypoint. The boat sends a delete request where the body is a JSON array of uuids of the completed commands.

/api/data.cgi

The data endpoint exists to store sample data from the boat, and serve it for the SPA to display.

POST

Used by the boat to record sensor data. The body of the request is a JSON object containing key values pairs representing sensor data for a given position and time.

GET

Retrieves a complete list of sample data received from the boat. Takes an optional timestamp parameter to limit the results to values after that point in time.

Frontend

Instead of generating a webpages for the interface with a CGI script, a SPA was written using the Angular platform. This generates a set of static files that replace index.cgi and commands.cgi of the old set up. Angular was used over the other frontend frameworks due to its popularity in the professional community at the time, and also because previous experience with predecessor framework AngularJS was held. The Angular CLI was used to initialize the project, generate template components and services as well as build the project for deployment to the server. The built project was then placed in the root of the 'solarboat' directory being served by the web server.

Design

Figure 6 outlines my initial design regarding the set of Angular components and services that would be need to be created for working application. Figure 7 shows the set of services and components as the application stands now.

Services

A local step was to create services to mirror the data on each endpoint on the server.
The "Sample Service" retrieves the existing position and sample data from the /api/data.cgi then polls it for new data, exploiting the 'fromTimestamp' optional parameter described above to minimize wasteful network traffic. The Sample service exposes an observable for sample data, which is explained in the Libraries section below.
The "Command Service" manages the local representation of the command data on the server from /api/command.cgi. It exposes methods for getting a copy of the command data as well as setting replacement data on the server.

The “Map Service” manages the state of the map so that position and zoom level is maintained across the various components as they are created and destroyed.

Components

To be able to view the path the boat had taken, a dedicated ‘Journey Component’ exists to contain a map which would display the between the positions recorded on the server. The map is updated in near real time as sample data is received by the server (delayed by up to the polling period of the sample service). Vertices on the path of the boat can be moused over for the complete sample data.

For planning boat missions there is a ‘Plan Component’ that also features a map. The map on the planning component features a similar line to that of the journey component, except that it outlines the currently planned route on the server. The map features menu items for appending to, modifying or clearing the waypoints/commands that the boat with obey.

The Sensor Component is intended to display graphs of sensor readings over time, especially useful for monitoring power levels.

Finally the Status Component was intended to just show the current status of the boat, that is position and sensor data, as well as any other information useful to the operator.

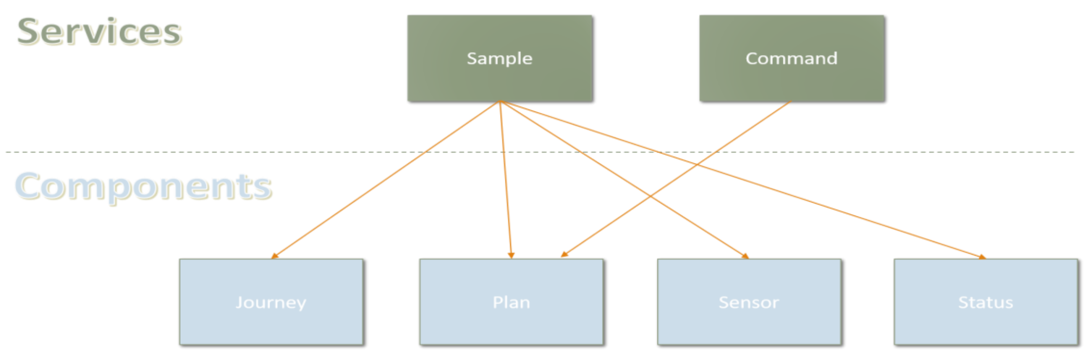


Figure 6: Inital Design of Angular SPA

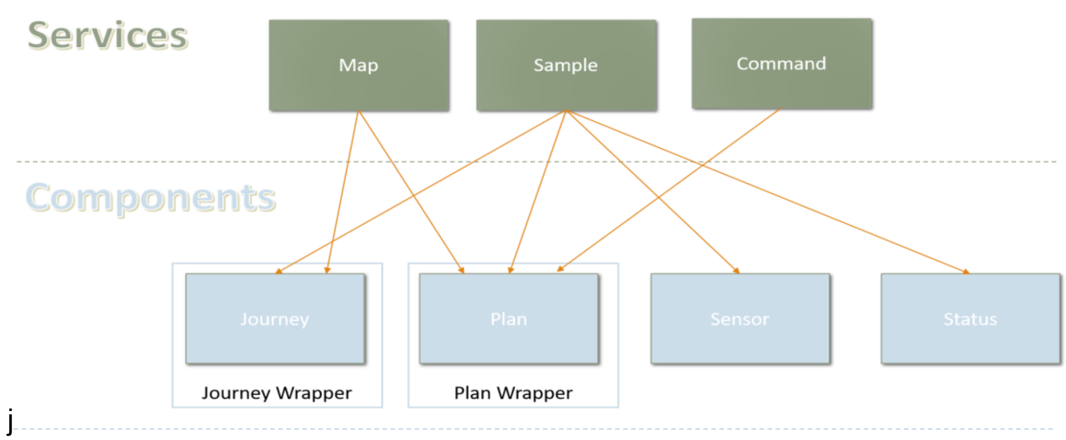


Figure 7: Resulting Design of Angular SPA

Libraries and Modules

The NgModule Angular Router [15] was used to allow for page like navigation between the major components listed above in a page like fashion.

The map featured in the Journey and Plan components uses Leaflet, considered the leading open source tool for maps in web development [38]. To support drawing on these maps, the plugin “Leaflet Draw“[39] was used. Both Leaflet and Leaflet Draw feature dedicated projects to assist in integrating them with Angular [40] [41], these allowed declaration of map features in the HTML templates of Components, through the use of Angular directives. Leaflet Toolbar [42] was used to have menus placed on the map. DyGraphs is a charting library designed to handle dense datasets [45], it is used for the plots in the Sensor component.

Angular itself makes use of a library called RxJS [43], which is a tool for writing code that is reactive. It does this through the use of ‘observables’, which are objects that emit events and can be ‘subscribed’ to. Angular uses observables for its HTTP client, to wrap events from UI elements and more. RxJS classes wrap our data from /api/data.cgi as a series of events in the Sample service. The sample service exposes a special kind of observable called a ‘ReplaySubject’ which is configured to replay all the data that it has emitted so far upon subscription. The command service exposes a ‘BehaviorSubject’ which behaves the same as an observable except that it keeps track of the most recent value and emits it to new subscribers, in this case the value is the most recent set of commands for the boat.

Autonomous Boat

The basic changes that needed to be made to the boat are simply the endpoints that it retrieves commands from, and sends position/sample data to. Modifications to how the boat sends the data, and processes commands were to be made also.

Results

The Single Page Application is functional. The boat’s ‘live’ journey and sensor readings are viewable on the relevant components and the waypoints can be edited by moving them around on a map, rather than by entering coordinates manually. Unfortunately time constraints prevented the incorporation of AIS data into the maps, data export is not yet built into the application and limiting the data in the application to a specific time range is currently not implemented. The API endpoints also don’t require authentication at present. Over the duration of this semester, there has not been reliable communication between the boat and the server, therefore the system has not been tested end to end. The round trip of Rottnest Island was not attempted.

Discussion

Difficulties and constraints

The has been a two major of difficulties encountered when attempting to test the system.
The first has been the reliability of the communication between boat and server.
Despite data transmission between boat and server being demonstrated to work reliably by Aaron Goldsworthy in his thesis [3], we have had very little success in replicating it. Attempts to send/receive data over HTTP with the old server setup and the new has resulted in very few successful transmissions (admittedly the majority of the effort in testing this was carried out by Wesley Coleman, not myself). The cause of this instability has eluded debugging efforts so far, several modems were tested as well as the structure of the HTTP requests sent. The server endpoints were demonstrated to work reliably using Postman [44] (a tool for assisting in the development of APIs). Because of this I believe the problem lies is either how the requests are being communicated to the modem, or that something is going wrong in the data transmission process.

The second of these has been how fragile the system has been during assembly. Wesley Coleman and I took the boat down to JH Abrahams Reserve on the Swan River on the 27th of April. After much preparation, the boat lost power upon being lowered into the water. The cause turned out to be that power connectors were very easily coming loose when the wooden planks hosting the electronics were placed in the hull. Wesley Coleman has since put in a significant amount of effort re-organizing and rewiring connections to address this, but in addition to his changes to the motor mounts to the boat, we ran out of time to test the boat further.

A major mistake encountered was spending time developing two different versions of backend server software to replace the existing CGI script setup. I wanted to use this project to learn new technologies and to this effect I decided to learn to develop a Python/Flask server over the summer break. Just before the start of the second semester of the project I decided that the solution I had come up with wasn't good enough and resolved to develop a Node.js/Express server instead as I knew I could quickly produce a quality solution. Several weeks into the second half of the project, It was pointed out that that we were using shared hosting for our server. Meaning that we don't have much control over the software running on the server, or the ports that were usable. While I managed to install the software required to get the Node.js/Express server running, due to limited privileges on the server, I was unable to forward requests to my software. Therefore I had to abandon my efforts and return to using CGI scripts as the mechanism for dynamic content. This was entirely my fault as in my proposal didn't include this idea in my proposal, but It did mean that a significant amount of time was consumed.

Future Work

Incomplete Work

Due to time constraints there are a number of features that have yet to make it into the current implementation of the SPA.

- Data Export: All the sensor and position data is already retrievable in JSON format by issuing a GET request to /api/data.cgi. Exporting this data as a CSV or another format might be convenient for future operators.
- Photo API: A Raspberry Pi camera was integrated onto the boat recently. An API for file/photo uploads would be nice to have, and the ability to view them in a gallery interface in the SPA, or on the map. Photos can simply be encoded as Base64 strings and sent via the same mechanism as the other API endpoints, or future work could be carried out to support multipart content. The Node.js/express server I worked on had the latter setup.
- Better display of sensor data on the map. A graphical/color coded representation of temperature or other data on the map would be superior to the current ‘popups’ showing sensor data.
- A component for selecting the time period of data is displayed to the user. This can simply trigger a refresh of the map in a component with filtered data, or it can be handled in the sample service itself.
- Securing the API endpoints: Currently any entity can enter data into the system if they know how to use the API. It would be smart to protect against outside interference.

Extending work

Suggestions for future work:

- Extending the autonomous behavior of the boat. Supporting different kinds ‘actions’ in the commands could yield interesting results. The ability to drift with the current/wind or hold a position for a duration would be nice features. This would require the ability to edit the action field of a command on the SPA.
    - Having the boat move according to its sensor readings, like a ‘Heat Seeking’ mode.
- Different Hull designs for the boat
- Should the hosting of the project ever change, replacing the CGI Script backend with a more modern alternative.
- Provided the communication method becomes more stable, including the recent standard output of the software running on the boat on the Status Component for diagnostics would be helpful for future development.
- A mechanism for the boat to update its software remotely

Conclusion

The aim of this thesis was to produce a Single Page Application that could be used to monitor and control the Autonomous Solar Boat at sea. Time constraints and technical issues have prevented proper testing of the solution with the boat itself, but during the development of the solution the behavior of the Server and SPA together is as expected. Improvements can be made to the way that data is displayed to the user and more features added (such as a photo API and gallery) however the basic functionality requirements were met. Primarily the command data is manipulatable on the map interface of the SPA, and position and sensor data over time is viewable.

The repositories of code relevant to this project can be found at:
https://github.com/MorganTrench/spab-cgi
https://github.com/MorganTrench/SPAB-Web
https://github.com/21485619/pyspar

References

[1] J. Hodge, “Project report: Autonomous solar powered boat,” Master’s thesis, University of Western Australia, 2017.

[2] J. Borella, “Solar powered autonomous boat (spab) thesis proposal,” Master’s thesis, University of Western Australia, 2017.

[3] A. Goldsworthy, “Remote control of Autonomous Surface Vessels”, Master’s thesis, University of Western Australia, 2018.

[4] M. Wasson, "Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET", *MSDN*, 2013. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx. [Accessed: 19- May- 2019].

[5] A. Matos, R. Verma and S. Masne, "What is REST – Learn to create timeless RESTful APIs.", *Restfulapi.net*, 2019. [Online]. Available: https://restfulapi.net/. [Accessed: 19- May- 2019].

[6] "URL Standard", *Url.spec.whatwg.org*, 2019. [Online]. Available: https://url.spec.whatwg.org/. [Accessed: 19- May- 2019].

[7] "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", *ietf.org*, 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230. [Accessed: 19- May- 2019].

[8] "Extensible Markup Language (XML) 1.1 (Second Edition)", 2006. [Online]. Available: https://www.w3.org/TR/2006/REC-xml11-20060816/. [Accessed: 19- May- 2019].

[9]"RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format", *Tools.ietf.org*, 2017. [Online]. Available: https://tools.ietf.org/html/rfc8259. [Accessed: 19- May- 2019].

[10] "Stack Overflow Developer Survey 2018", *Stack Overflow*, 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018. [Accessed: 19- May- 2019].

[11] "Stack Overflow Developer Survey 2019", *Stack Overflow*, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019. [Accessed: 19- May- 2019].

[12] "Angular", *Angular.io*, 2019. [Online]. Available: https://angular.io/docs. [Accessed: 19- May- 2019].

[13] "Native mobile apps with Angular, Vue.js, TypeScript, JavaScript - NativeScript", *NativeScript.org*, 2019. [Online]. Available: https://www.nativescript.org. [Accessed: 19- May- 2019].

[14] "Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS.", *Electronjs.org*, 2019. [Online]. Available: https://electronjs.org. [Accessed: 19- May- 2019].

[15] "Angular", *Angular.io*, 2019. [Online]. Available: https://angular.io/api/router/Router. [Accessed: 19- May- 2019].

[16] "RFC 3875 - The Common Gateway Interface (CGI) Version 1.1", *Tools.ietf.org*, 2004. [Online]. Available: https://tools.ietf.org/html/rfc3875. [Accessed: 19- May- 2019].

[17] F. Boender, "Apache, FastCGI and Python", *Electricmonk.nl*, 2019. [Online]. Available: https://www.electricmonk.nl/docs/apache_fastcgi_python/apache_fastcgi_python.html. [Accessed: 19- May- 2019].

[18] “PEP 3333 -- Python Web Server Gateway Interface v1.0.1", *Python.org*, 2010. [Online]. Available: https://www.python.org/dev/peps/pep-3333/. [Accessed: 19- May- 2019].

[19] "Reverse Proxy Guide - Apache HTTP Server Version 2.4", *Httpd.apache.org*, 2019. [Online]. Available: https://httpd.apache.org/docs/2.4/howto/reverse_proxy.html. [Accessed: 19- May- 2019].
[20] Electric Cars, Solar Panels & Clean Energy Storage | Tesla", *Tesla.com*, 2019. [Online]. Available: https://www.tesla.com/. [Accessed: 19- May- 2019].
[21] M. Matousek, "10 electric cars that will challenge Tesla's Model 3", *Business Insider Australia*, 2018. [Online]. Available: https://www.businessinsider.com.au/electric-cars-challenging-tesla-model-3-2018-1?r=US&IR=T. [Accessed: 19- May- 2019].
[22] F. Moogal, "Tesla Autonomy Day: What We Learned | CleanTechnica", *CleanTechnica*, 2019. [Online]. Available: https://cleantechnica.com/2019/04/23/tesla-autonomy-day-what-we-learned/. [Accessed: 19- May- 2019].
[23] A. delivery, "Amazon and UPS are betting big on drone delivery", Business Insider, 2018. [Online]. Available: https://www.businessinsider.com/amazon-and-ups-are-betting-big-on-drone-delivery-2018-3/?r=AU&IR=T. [Accessed: 13- Sep- 2018].
[24] [7]"Canberra – Wing", *Wing*, 2019. [Online]. Available: https://wing.com/intl/en_au/australia/canberra/. [Accessed: 19- May- 2019].
[25] "WA Police plan to test drones for spying on criminals", The West Australian, 2018. [Online]. Available: https://thewest.com.au/news/wa/wa-police-plan-to-test-drones-for-spying-on-criminals-ng-b88478839z. [Accessed: 13- Sep- 2018].
[26] F. Plumet, C. Petres, M. Romero-Ramirez, B. Gas and S. Ieng, "Toward an Autonomous Sailing Boat", IEEE Journal of Oceanic Engineering, vol. 40, no. 2, pp. 397-407, 2015.
[27]D. Eickstedt, M. Benjamin, H. Schmidt and J. Leonard, "Adaptive Control of Heterogeneous Marine Sensor Platforms in an Autonomous Sensor Network", 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006.
[28] F. Plumet, C. Petres, M. Romero-Ramirez, B. Gas and S. Ieng, "Toward an Autonomous Sailing Boat", *IEEE Journal of Oceanic Engineering*, vol. 40, no. 2, pp. 397-407, 2015.
[29] "The Microtransat Challenge", *Microtransat.org*, 2019. [Online]. Available: https://www.microtransat.org/. [Accessed: 19- May- 2019].
[30] "The Microtransat Challenge", *Microtransat.org*. [Online]. Available: https://www.microtransat.org/2018teams.php. [Accessed: 19- May- 2019].
[31] "SeaCharger Oceangoing Autonomous Boat - Home", SEACHARGER, 2018. [Online]. Available: http://www.seacharger.com/about.html. [Accessed: 13- Sep- 2018].
[32]"SeaCharger Oceangoing Autonomous Boat - Home", SEACHARGER, 2018. [Online]. Available: http://www.seacharger.com/tracking.html. [Accessed: 13- Sep- 2018].
[33] "Angular", *Angular.io*, 2019. [Online]. Available: https://angular.io/cli. [Accessed: 19- May- 2019].
[34] "Angular Material", *Material.angular.io*, 2019. [Online]. Available: https://material.angular.io/. [Accessed: 19- May- 2019].
[35] "React Native · A framework for building native apps using React", *Facebook.github.io*, 2019. [Online]. Available: https://facebook.github.io/react-native/. [Accessed: 19- May- 2019].
[36] "Vue.js", *Vuejs.org*, 2019. [Online]. Available: https://vuejs.org/. [Accessed: 19- May- 2019].
[37] "RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace", *Tools.ietf.org*, 2005. [Online]. Available: https://tools.ietf.org/html/rfc4122. [Accessed: 19- May- 2019].

[38] "Leaflet/Leaflet", *GitHub*, 2019. [Online]. Available: https://github.com/Leaflet/Leaflet. [Accessed: 19- May- 2019].
[39] "Leaflet/Leaflet.draw", *GitHub*, 2019. [Online]. Available: https://github.com/Leaflet/Leaflet.draw. [Accessed: 19- May- 2019].
[40] "Asymmetrik/ngx-leaflet", *GitHub*, 2019. [Online]. Available: https://github.com/Asymmetrik/ngx-leaflet. [Accessed: 19- May- 2019].
[41] "Asymmetrik/ngx-leaflet-draw", *GitHub*, 2019. [Online]. Available: https://github.com/Asymmetrik/ngx-leaflet-draw. [Accessed: 19- May- 2019].
[42] [7]"Leaflet/Leaflet.toolbar", *GitHub*, 2019. [Online]. Available: https://github.com/Leaflet/Leaflet.toolbar. [Accessed: 19- May- 2019].
[43] [8]"ReactiveX/rxjs", *GitHub*, 2019. [Online]. Available: https://github.com/ReactiveX/rxjs. [Accessed: 19- May- 2019].
[44] [9]"Postman | API Development Environment", *Postman*, 2019. [Online]. Available: https://www.getpostman.com/. [Accessed: 19- May- 2019].
[45]"DyGraphs", *Dygraphs.com*, 2019. [Online]. Available: http://dygraphs.com/. [Accessed: 19- May- 2019].

APPENDICES

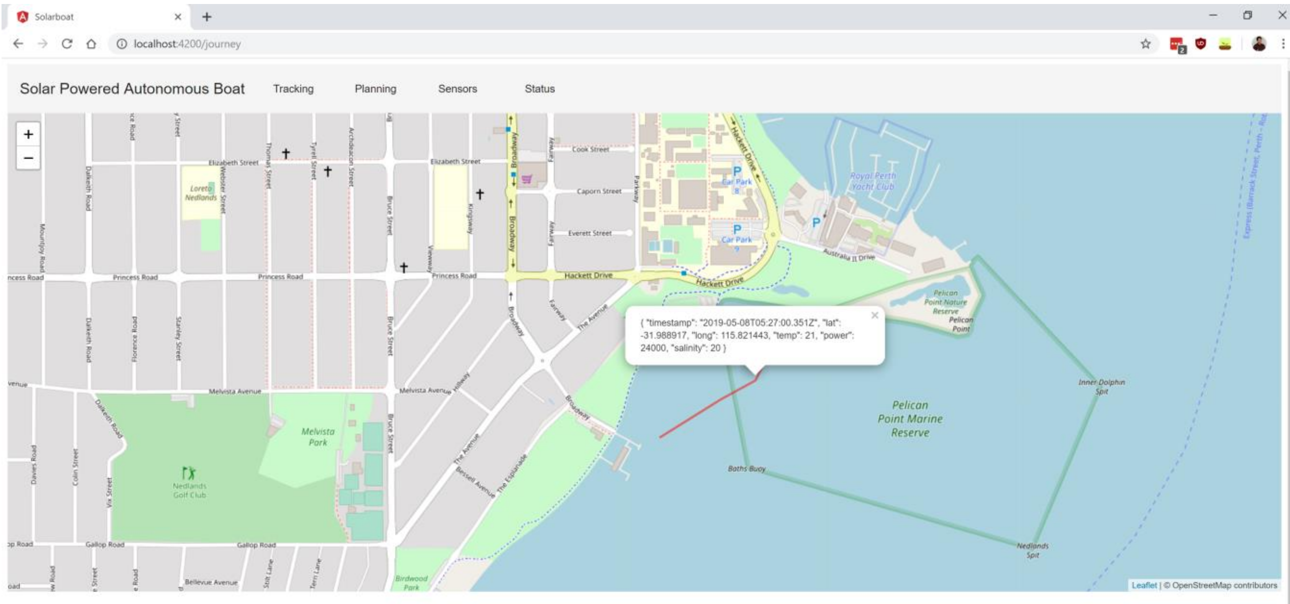


Figure 8: Journey Component with mouseover of a datapoint

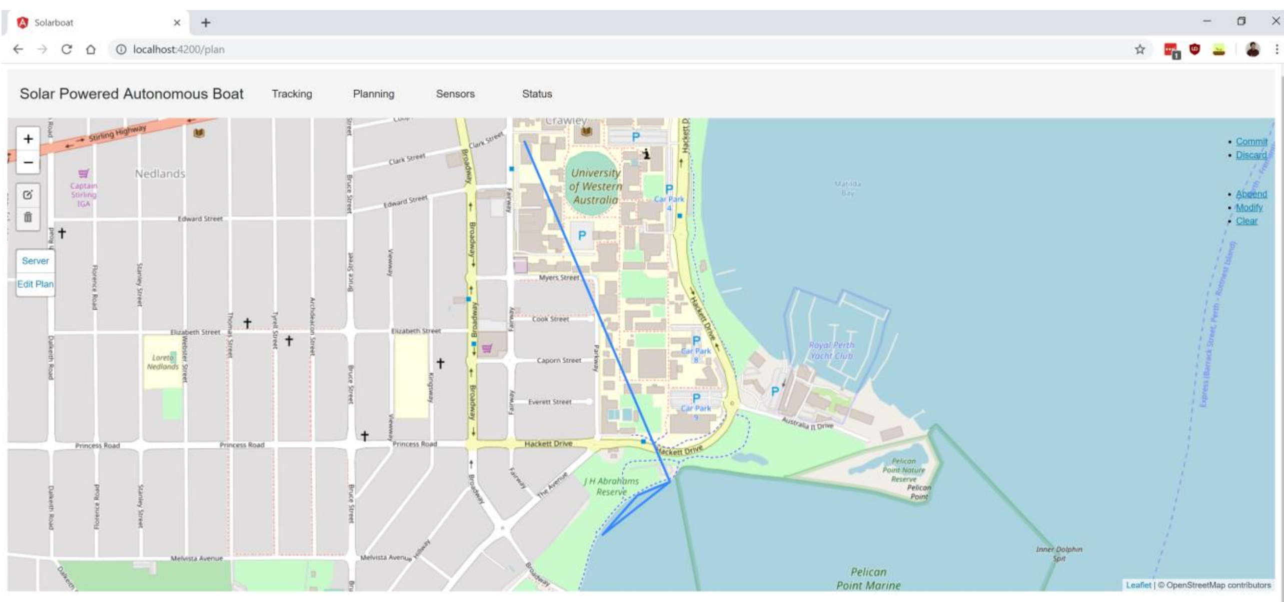


Figure 9: Plan Component - Default

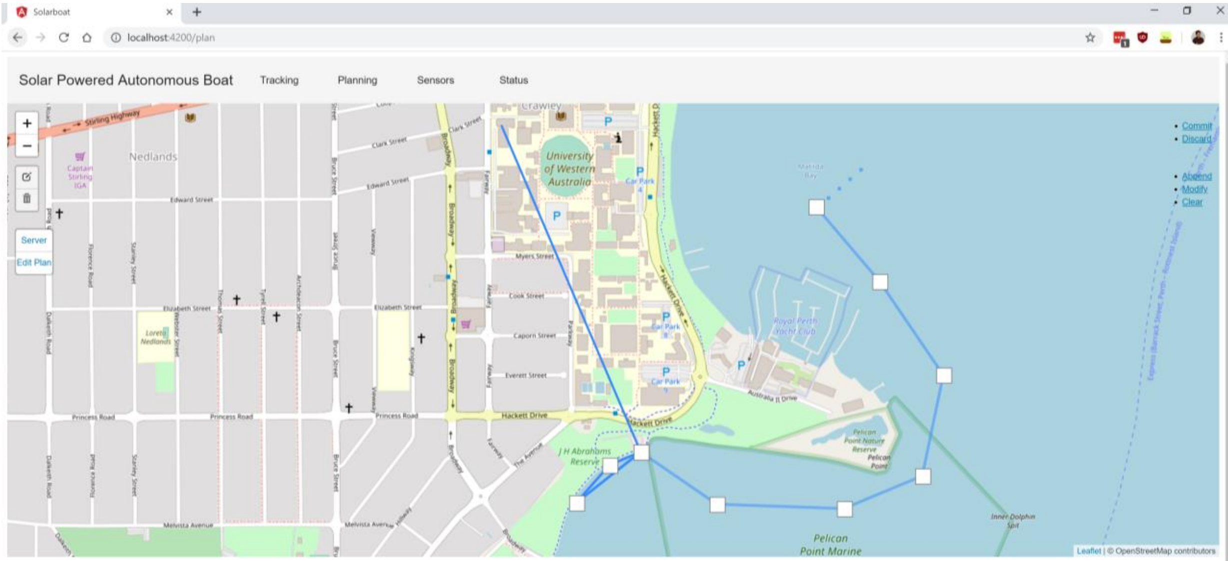


Figure 10: Plan Component - Appending to route

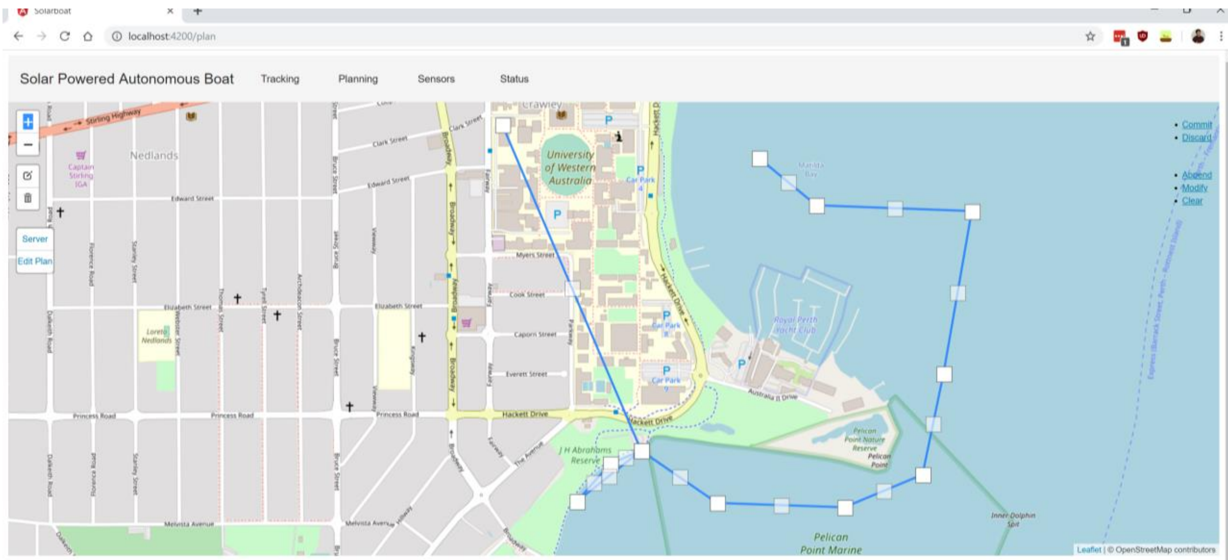


Figure 11: Plan Component - Editing existing route

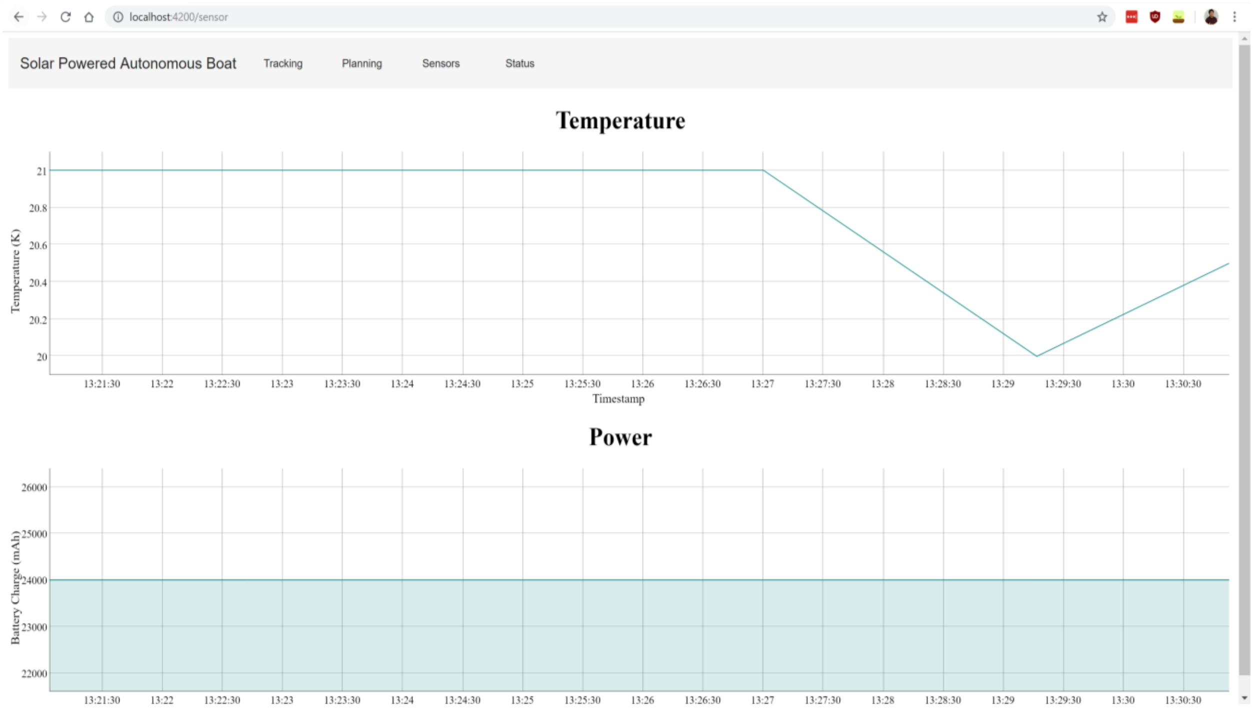
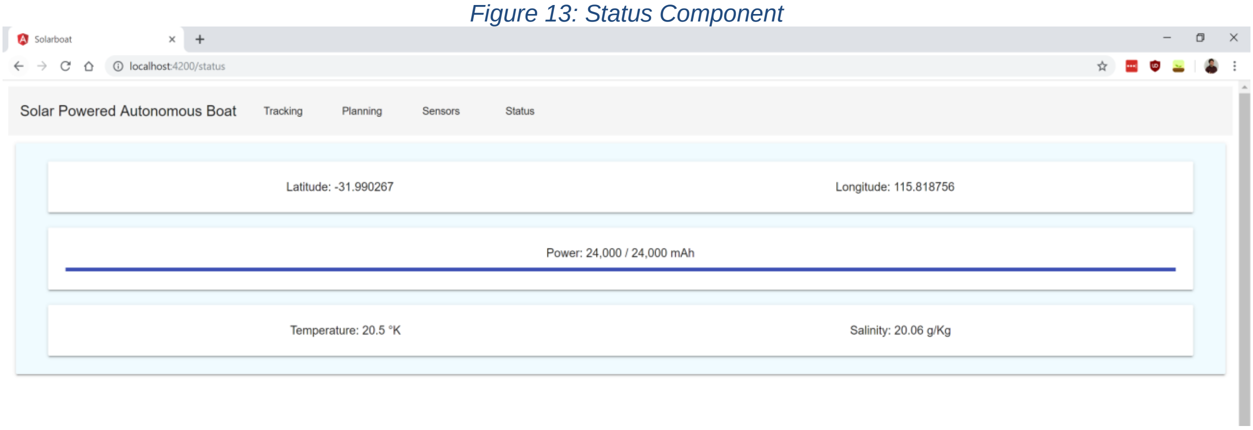


Figure 12: Sensor Component - Temperature and Power Graphs (power data is dummy data at present)

Figure 13: Status Component



GET

/api/data.cgi?fromTimestamp=<unixTime>


```
[
    {
        "temperature": 22.25,
        "sourceId": "spar1",
        "timestamp": 666625,
        "longitude": 115.8160405,
        "salinity": 0,
        "latitude": -31.9791509,
        "id": 15
    },
    {
        "temperature": 22.125,
        "sourceId": "spar1",
        "timestamp": 249907,
        "longitude": 115.8160599,
        "salinity": 0,
        "latitude": -31.9795484,
        "id": 9
    },
    {
        "temperature": 23.25,
        "sourceId": "spar1",
        "timestamp": 191577,
        "longitude": 0,
        "salinity": 0,
        "latitude": 0,
        "id": 11
    },
    {
        "temperature": 23.5,
        "sourceId": "spar1",
        "timestamp": 128638,
        "longitude": 0,
        "salinity": 0,
        "latitude": 0,
        "id": 10
    },
    {
        "temperature": 22.187,
```


POST

Requires the type key


```
{
        "temperature": 22.25,
        "sourceId": "SPAB",
        "timestamp": 1666625,
        "longitude": 115.8160405,
        "salinity": 0,
        "latitude": -31.9791509
}
```

```
{
    "message": "Inserted successfully",
    "type": "dataUpload",
    "success": true
}
```


Figure 14: Example Requests and Responses to the /api/data.cgi endpoint

PUT


```json
[
    {
        "action": "moveTo",
        "latitude": -31.987339,
        "longitude": 115.820166,
        "duration": 0
    },
    {
        "action": "moveTo",
        "latitude": -31.987631,
        "longitude": 115.819212,
        "duration": 0
    },
    {
        "action": "moveTo",
        "latitude": -31.988602,
        "longitude": 115.818255,
        "duration": 0
    },
    {
        "action": "moveTo",
        "latitude": -31.987339,
        "longitude": 115.820166,
        "duration": 0
    }
]
```


GET

As the boat uses this endpoint, a way to identify the request is required.

Using a "type" key in the object response


```json
{
    "data": [
        {
            "uuid": "57a651f4-5afb-4aea-87b2-d141dea9d179",
            "duration": 0,
            "longitude": 115.820166,
            "latitude": -31.987339,
            "action": "moveTo",
            "id": 1
        },
        {
            "uuid": "1d29da82-65a3-4459-bb89-361d6b35d29a",
            "duration": 0,
            "longitude": 115.819212,
            "latitude": -31.987631,
            "action": "moveTo",
            "id": 2
        },
        {
            "uuid": "e801a67c-8cfc-4bd3-85e6-3c3e3804c73c",
            "duration": 0,
            "longitude": 115.818255,
            "latitude": -31.988602,
            "action": "moveTo",
            "id": 3
        },
        {
            "uuid": "a06f8482-64be-4dcc-80cc-42c351d7201d",
            "duration": 0,
            "longitude": 115.820166,
            "latitude": -31.987339,
            "action": "moveTo",
            "id": 4
        }
    ],
    "type": "cmdList"
}
```


DELETE

Requires the type key


```json
[
    "3b255198-f9db-40db-b462-617fc4133e00",
    "b54ba8ea-8a56-4a34-bdd4-b549489706b3",
    "6b01eb36-d5f9-4c5c-b48e-1756f01fa0ab",
    "2ec78f55-9686-4064-aeab-49f99d215f05"
]
```

```json
{
    "message": "Commands completed successfully",
    "type": "cmdComplete",
    "success": true
}
```


Figure 15: Example Requests and Responses to /api/command.cgi