Final Report for: Scalable Web Application Framework for Monitoring Energy Usage on Campus

Daniel Schroeder, Aubrey Thenell, Parker Bruni
Group 57

Abstract
The purpose of this document is to provide final documentation for Group fifty-seven's capstone project.

CONTENTS

1 Introduction to Project

1.1 Why is this project important?

As the population of the world grows exponentially, the demand for energy grows with it. Unfortunately, a massive source of energy for humans comes from non-sustainable fuel burning. This burning releases greenhouse gasses into the earths atmosphere, which results in the heating of our planet and many consequences that come as a result. To combat this, communities have decided that it is time to get our energy from more sustainable sources, as well as use it in more sustainable ways. In order to be more sustainable and conscious of our energy consumption, we need to implement modern tools to monitor energy usage data and use that data to make informed decisions on future infrastructure

projects. This is necessary to reduce our carbon footprint, reduce costs, and move society to a more sustainable (and eventually fully sustainable) future in regards to energy consumption.

Oregon State University is revered for its energy efficiency and sustainability and is committed to reducing its carbon footprint through renovations and sustainable consideration with new projects. In this era of technology it is necessary to utilize the tools that are available to us when designing and implementing systems that will increase energy efficiency. Oregon State focuses on reducing its overall costs while supporting a more sustainable environment through less consumption and emissions. With this in mind, OSU has installed energy meters in campus buildings to monitor and record energy data so that it may be analysed by members of Oregon State Universitys Sustainability Office. The data that is gathered from these systems can be used to make educated decisions about the current energy usage systems in place, such as monitoring strange fluctuations or anomalies in energy usage in any given building. OSU can address and correct any potential waste from occurring as well as provide a foundation of knowledge to reference when planning future infrastructure projects by utilizing these systems.

1.2 Who requested it?

Oregon State University's Office of Sustainability previously had contracted a company Lucid to create a web application that acted as an administrative dashboard for monitoring energy use across campus. However, as Oregon State University Sustainability Office scales up its operations and automated reporting, the price of Lucids contract becomes exponentially expensive to maintain. In order to continue data aggregation efforts, the Oregon State Sustainability office has recognized an opportunity for our team to create and plan a project that can take the place of the legacy system. Through creating an EECS capstone project, the Office of Sustainability was able to have an internally sourced web application that can be modified to whichever specifications they desire. It will also save the University hundreds of thousands of dollars that was previously spent on Lucid's services and direct that money towards more sustainable projects for the entire University and help Oregon State reach its carbon neutrality goal of 2025.

1.3 Meet the team

1.3.1 Daniel Schroeder - Developer

Daniel was responsible for generating a lot of the front-end data binding implementations with AngularJS. He implement the majority of functionality for all of the main components of the application (blocks, dashboards, stories, buildings). A lot of this process involved finding intuitive ways to query for aggregated data and populate web forms or UI components according to what is needed. A majority of Daniel's contributions revolved around generating a lot of Bootstrap4 and AngularJS webpages that auto-populated drop-down menus, graph data, or user objects so the user could view, create, edit, and delete all of the major components of the application. Additionally, Daniel needed to write route handler functions for and Express.js app that aggregated certain datasets and returned the correct data to the client.

1.3.2 Aubrey Thenell - Developer

Aubrey was responsible for most of the backend based code. Specifically, she was responsible for the data collection server. This contribution allows data storage into the database, which is retrieved by the web application. She was also responsible for all email based functionality of the website. This includes invitation emails, high energy usage detection emails, and down meter emails. She was also responsible for a lot of the security based functionality. This includes all form validation to ensure malicious injection won't affect the database. It also includes sitewide permissions authentication. This ensures that people don't access things they shouldn't have access to. Aubrey also wrote the interface for the built in Kilowatt Crackdown Challenge that the Office of Sustainability runs annually. Finally, Aubrey contributed by documenting all of the code with JSDocs.

1.3.3 Parker Bruni - Developer

Parker was involved in designing the UI and overall look and feel of the application, as well as formatting many of the elements within the site. He used methods of CSS and Bootstrap as well as knowledge of what makes a web application

appealing to a user to give the site a clean and modern feel by use of color schemes and minor site animations. He also implemented the initial stages of charting using chart.js and implemented the google maps API that is found on the home page.

1.4 Role of the Client

Our client, Jack Woods, is a student at Oregon State University who is employed by the Office of Sustainability. He served as a direct line of communication between our development team and the Office of Sustainability. He worked very closely with the team on development having regular meetings and was involved in team discussions online. As well as communication, Jack has contributed to the project in other ways, such as setting up our EC2 instance so our site may be publicly accessible as well as working with the team on code to insure correct functionality of some components.

2 Requirements Document

2.1 Introduction

2.1.1 Purpose

The purpose of this document is to outline the project and all associated information. Furthermore, it will outline how the end product will be used, how it was developed, and all resources and documentation referenced and used during development. This document will also describe the applications target audience, user interface, and any hardware/software requirements. Finally, it will define how our client, team, and users will see the final product and related functionality.

2.1.2 Scope

This project will provide a web application that displays different collections of energy use reports for campus buildings. Each of these collections will be customizable, with the ability to add, remove, and filter building data according to user preference. The web application will allow users to visually comprehend energy data in a way that can raise awareness about consumption and reduce the campus-wide carbon footprint. The web application will also have three different permission levels: Guest, Registered Users, and Admin Users. Each one will provide permissions to what can be viewed and edited.

2.1.3 Definitions, acronyms, and abbreviations

Term	**Definition**
OSU	Oregon State University
MEAN stack	MongoDB, Express, AngularJS, Node.js
JSON	JavaScript Object Notation
AcquiSuite	Data Acquisition Servers responsible for metering energy usage
Collections	User Interface dashboard that displays assortment of blocks
Blocks	Charts/graphs of building data
Stories	Collections of dashboards

2.1.4 References

The references are:

- AcquiSuite and Modbus Information [1] [2] [3] [4]
- D3.js [5]
- Vis.js [6]
- The M.E.A.N. Stack [7]

- JSON [8]
- BuildingOS [9]
- Mocha.js [10]

2.1.5 Document Overview

The remaining sections of this document provide a more elaborate description of product functionality, assumptions, and specific requirements. Section 2 provides information about functional requirements, data requirements, and assumptions made for designing the *Scalable Web Application Framework for Monitoring Energy Usage on Campus*. Section 3 outlines the specific requirements for the final product, the external interfaces that communicate with the software, and the functional requirements of the system.

2.2 Overall description

This section will provide an overview of the web application as a whole, including:

1) Details about the user interface and expected user interactions.
2) An outline of specific constraints and dependences included in development.
3) Background information about the specific software requirements.

2.2.1 Product perspective

This product will replace the current web application that is used by The Oregon State Sustainability Office. The product will display energy usage information about Oregon State University buildings through an intuitive user interface. The application will gather data from energy meters by connecting to AcquiSuite^{TM}data acquisition servers. The energy data from the meters will be transferred to a database that the product will access directly. Users will interact with the application interface from an internet browser application.

2.2.2 Product functions

The product will allow users to create accounts that may be given either administrative or user permissions. Users have the ability to personalize their accounts. Administrators accounts will have special capabilities that user accounts will not.
The application will allow users to create customizable dashboards that will contain easily adjustable blocks of campus building data that will contain the campus building energy usage data. These blocks of data will be the basic building blocks for the dashboard and will provide an intuitive view of the data. They will feature various graph types, building energy efficiency rankings, and data trends.
Each OSU building that contains the energy monitoring meter(s) will have a specific, non-customizable page that will display general information. The product will also have a public interface and a private administrative interface.
Administrators of the application will have the ability to add, remove, or edit entire buildings profiles, building subspaces, or individual meters.

2.2.3 User characteristics

A user that will be using the general public UI will not need to know any specific information about the application to navigate the various energy data presentations. A public user will be able to intuitively navigate the UI at their discretion.
An administrative level user will need a basic understanding of the tools of the application because they will be allowed the freedom to control parts of the website as well as have access to more specific energy data within the application. An administrator will likely not need extensive training to use this application for more specific purposes as the administrator UI will be designed to be intuitive to navigate.

2.2.4 Constraints

Data updates will be limited to a granularity of 15 minute intervals. The data acquisition server is capable of providing a granularity of up to 15 second intervals but it is not necessary for the purposes of the application. Any other constraints on the product are subject to the internet browsing interface that is accessing our product. The UI will be intuitive and able to navigate data presentations, but will not allow manipulation of the data or underlying structure of the application.

2.2.5 Assumptions and Dependencies

The application will be dependant on the data acquisition server. If the data acquisition server is drastically changed or removed, the application will not be functional. Meter data within the application will also be limited by the functionality of the meters themselves. Should a meter malfunction, the energy data will not be gathered which may cause some of the data presentations to deviate from expected data. The use of the application will require a compatible web browsing interface, which may be limited to browsers such as Firefox, Chrome, and Internet Explorer. The functionality of the product will depend on those internet browsers performing as expected with regards to their intended functionality and internet access.

2.2.6 Apportioning of Requirements

Future versions of the application may include features such as cost tables, automated electronic invoice generation, energy billing analysis capabilities, budget analysis capabilities, and mobile energy data entry.

2.3 Specific Requirements

2.3.1 External Interfaces

AcquiSuite data acquisition servers [meters] made by Obvious

- Used for collecting electric, water, gas, steam, and other energy parameters over the web.
- Data is received through IP-based connection.
- Data can be reached anywhere with an internet connection, as long as the AcquiSuite is online.
- Data will be collected every 15 minutes.
- Data from the AcquiSuite servers will be stored into a database.

2.3.2 Functions

This section defines how the software system should behave with regards to input and output.

- The system shall receive and store data from AcquiSuite data acquisition servers into the database.
- The system shall have input validation measures in place to monitor incoming data and protect from malicious injections.
- The system shall retrieve data from the database and populate webpages with filtered datasets when prompted.
- The system shall have permission based restrictions for accessing certain data.
- The system shall generate alerts for offline buildings and high energy usage.
- The system shall generate emails for users to access a sign-up form and create an account.
- The system shall be able to create arbitrary combinations of datasets given user filters.
- The system shall be able to calculate rankings for all the buildings based on certain metrics like “kilowatt-hour consumption”.

2.3.3 Performance Requirements

This section describes the functionality requirements for the software as well as what a user should be able to accomplish when using the web application.

2.3.3.1 Users

The system will have 3 types of users: admin level users, registered users, and generic users. An administrative user will be able to add new meters, buildings, and other objects into the database. A registered user, for example the head of a department, can create their own stories with information and collections that are unique to their own interests or departments. Lastly, a generic user can simply view the publically facing web application and see public dashboards and browse public stories.

- A registered user should be able to customize dashboard layouts in a grid-based orientation. A *story* page should have a customizable layout where a user can add different *blocks* with information relevant to their personal needs.
- An administrative user should be able to add buildings to the database through a web form.
- An administrative user should be able to add data aquisitions servers.
- An administrative user should be able to download specific datasets as a .csv file.
- An administrative user should be able to customize public stories.

2.3.3.2 Data Management

- The web application should update data blocks every 15 minutes as new data is received into the database.
- The web application should allow users to create which are collections of dashboards. These *stories* are meant to bring related buildings and datasets together into intuitive groups, for instance “Residence Halls” or “Engineering Buildings.”
- The application should be able to filter building data by date range specifications.
- The web application should be able to scale up to as many buildings are on campus.
- The web application should properly create and store new entities (buildings, users, meters) into the database.

2.3.3.3 Visualization

- The web application should be able to generate different types of graphs for different *blocks*. For example load-profile charts, comparative line charts, and heat maps.
- The web application should have generic pages for each building in the database. This page will display a series of graphs and charts that outline energy usage for a particular building.
- The *blocks* on each page should automatically update as new data is received by the database.

2.3.3.4 Objects

The system should have three main entities:

- **Building** - A building on campus that is connected to a data acquisition server.
- **Meter** - A specific data acquisition server.
- **User** - A user with a specific role.

2.3.4 Logical Database Requirements

Most of the calls to the database will be requests for meter data from a specific building or group of buildings. Access to the database should be hidden from the user and only accessed from the back-end of the application itself. Any input into the database should be validated and encrypted, if applicable.

2.3.5 Design Constraints

Design constraints may include server availability which could harm scalability.

2.3.5.1 Standards Compliance

There may be standards when storing energy data based on The Office of Sustainability Standards. [Check with Client]

2.3.6 Software System Attributes

2.3.6.1 Reliability

The system will be reliable at the time of deployment if all data displayed in graphs and charts is correct.

2.3.6.2 Availability

The system should be consistently available as long as the servers are up and running. As a web application, the system will be available via URL.

2.3.6.3 Security

The system will validate input for duplicate and malicious content. The system will also have back-end functionality to protect user passwords through proper hashing. The web application will restrict access to certain pages and content based on user roles and permissions.

2.3.6.4 Maintainability

Maintainability for this application should be simple for anyone with web development experience. The application code will be organized into logical directories and sub directories that should mimic a full stack web application. There will be documentation that explains how to connect to the AcquiSuite servers and how to use our API for data collection.

2.3.6.5 Portability

Our web application should be visible and accessible from most web browsers. There are not any host-dependent constraints since it will be hosted from a central server and available to the public via the internet. The source code should stay on Github so modifications can be made easily.

include the original document, showing what you thought, at the time, was the project definition with the original Gantt chart Add (your client should have okay'd): What new requirements were added? What existing requirements were changed? What existing requirements were deleted? Why? Add: Final Gantt Chart as a record of what happened when.

3 DESIGN DOCUMENT

3.1 Introduction

3.1.1 Purpose

The purpose of this Software Design Document (SDD) is to provide details about the architecture of the web application as well as details about each component. It will describe the underlying design of each component and their purpose within the context of the application that will allow Oregon State affiliates to effectively monitor energy usage on campus.

3.1.2 Scope

The software outlined in this document will function as an interface by which Oregon State students, faculty, and affiliates may monitor energy usage of various buildings located on the Oregon State Campus. It will act as a tool for users to make informed infrastructure decisions and adjustments or act as a display piece to be presented within buildings on campus. It will serve the OSU Office of Sustainability as a replacement to an outdated and costly implementation.

3.1.3 Overview

Oregon State University is constantly making strides to reduce its carbon footprint and reduce its energy consumption. There is a carbon neutrality goal for 2025 where the university is trying to generate as much energy as it uses and have a net carbon footprint of zero. Our web application aims to monitor the energy use of buildings on campus in order to create a visual representation of each building's consumption and incentivize people to monitor their consumption habits and reduce the campus's overall consumption.

Our web application will resemble an administrative dashboard with charts and graphs of energy use over time for individual buildings and subgroups of buildings on Oregon State's campus. Our application will have a series of public facing pages that will show energy usage for all the buildings on campus that have AcquiSuite meters as well as display user generated stories of personalized dashboards. Each page will be a grid-based dashboard with personalized blocks for displaying data through time-series charts or graphs. A user will be able to add different blocks to create unique dashboards for their own unique interests and subsets of buildings. An example of this would be a dashboard designated to only residence halls where each block shows a usage over time graph for energy consumption of each residence hall. Users will be able to apply date filters to their blocks to generate different data sets and more explicit visualizations.

Our web application will be constructed using a MEAN Stack framework hosted on AWS. Our application will have a MongoDB database server and a Node.js application server hosted on a single virtual EC2 instance to make deployment easy and reliable.

3.1.4 Reference Material

- Bootstrap4 Dashboards [11]
- Using OAuth 2.0 to Access Google APIs [12]
- Data-Driven Documents [13]
- AngularJS [14]
- AcquiSuite by Obvius [4]

3.1.5 Definitions and Acronyms

This subsection will define terms and acronyms that are specific to the application that may otherwise be misunderstood or poorly interpreted.

- **Dashboard:** A grid-based information management tool for visually tracking and displaying metrics and data through graphs and charts.
- **Block:** An individual graph or chart depicting time-series data. More specifically, it is an HTML element containing one or more buildings, with filter buttons, informative text, and graphs.
- **Story:** A user generated collection of dashboards. What each story displays is based solely on user preferences. With multiple stories, the user can select between "presets" that can show different dashboards.
- **MEAN Stack:** An acronym used to define a full stack application engineered from the MongoDB, Express.js, Angular.js, and Node.js frameworks.
- **Bootstrap:** Bootstrap CSS is a front-end framework that uses component templates to easily generate different HTML elements like buttons, navigation, or forms.
- **OAuth 2.0:** OAuth 2.0 is an authorization protocol that grants authentication through tokens rather than credentials.
- **Filter:** Filters are essentially parameters for fetching data from that constrain the subset of data being received to the specifications of the filter parameters (i.e. a date range).
- **D3:** D3.js is a visualization framework that appends charts and graphs to DOM elements on a webpage.
- **AcquiSuite:** AcquiSuites are data acquisition servers made by a company called Obvius that post building meter data to a designated IP address.
- **AWS:** AWS is an acronym for Amazon Web Services which offers reliable cloud computing services for building and hosting web applications.
- **Passport:** Passport is authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, etc.

3.2 System Overview

Our web application will provide users with the functionality to create unique dashboard-style collections of charts and graphs to analyze energy use from Oregon State's campus buildings. Our system will be highly modular, containing multiple smaller components that provide their own individual functionality while sharing information across other components. Designing a modular architecture will simplify the code base and allow the developers to encapsulate specific data models as their own entities and keep abstractions in the system differentiable.

Our design also stems from the main constructs behind AngularJS and data injection. Most of our components and subsystems are, at large, data models and objects that hold all the necessary information to create graph and charts. With this design, AngularJS services will extract data from the session user and rendered components and generate render the desired graphs and views to the content container. This allows the application to break up back-end services as well as UI components to simplify how data is gathered and processed.

3.3 Design Viewpoints and Concerns

3.3.1 Stakeholders and Concerns

Developers concerns consist of simplifying complex model functions, replicable components, modular subsystems, and simple data access functionality. Addressing and capitalizing on these concerns will create an application design that will enhanced understanding, simplify implementation, and produce a high-end product. From a developer's standpoint, it is also essential to design an application that is ideal for both implementation and usability.

Client concerns revolve around satisfying the requirements of the application and providing all the necessary user functionality. The implementation should adhere to the client's requests and specifications while creating a user-friendly interface to perform all the necessary actions.

3.3.2 Design Viewpoints

3.3.2.1 Context Viewpoint

The design of this web application will greatly depend upon satisfying the user actions outlined in the requirements document. With that in mind, all components and subsystems should cater towards simplifying the user's experience while performing actions and navigating through the application. This viewpoint provides design constraints for individual component functionality, as well as the overall layout and structure of the application as a whole. Our design implementations should cater to a wide range of users while creating an elegant way for the user to create, edit, and remove content.

3.3.2.2 Composition Viewpoint

This application contains a number of components that are recurrently used throughout the application and across users. These include the basic data blocks, dashboards, and stories that a user can create. The design of this application should embrace the modularity of these components and provide a consistent model for individual functionality while maintaining the ability to communicate and pass data across different components or objects.

3.3.2.3 Dependency Viewpoint

With a design focused around modularity and individualized components, there needs to be designs for services and dependencies so each component can interact. Some examples of these are: how a block component depends on multiple entities contained inside the block object like the building objects stored within the block, the filter's current values, the graph template, and the graphing service. The design of this application should encapsulate the relationships between different modular components and provide intuitive means for inter-component communication.

3.3.2.4 Information Viewpoint

Our applications design should provide logical ways of storing data and receiving data from building meters. This involves an explicit declaration of individual data models and how how data is created and stored. Our design will attempt to create a modular data structure where data specific to certain entities reside in the entity objects themselves. This allows the application to access data from an individual component as it gets created or moved throughout the application.

3.3.2.5 Logical Viewpoint

The application should maintain a consistent model for each component which can be used in different ways for different purposes. An entity should contain and provide the necessary information as it is needed throughout various areas of the application. The design of the application as a whole should consider where certain data is needed, and what data models must be present to satisfy the system's requests.

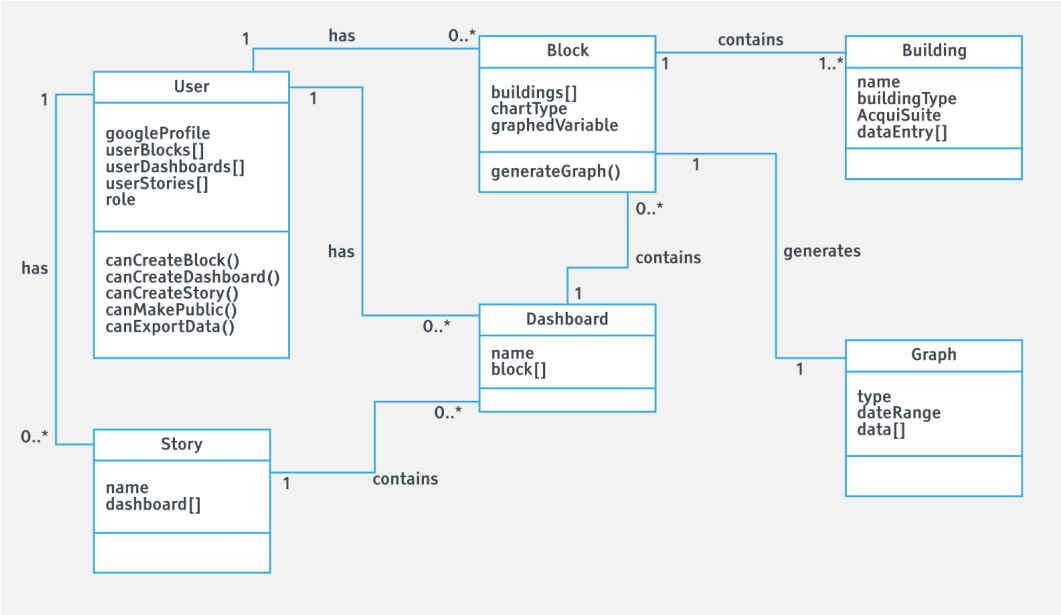


Figure 1: A UML diagram to show component relationships.

3.3.2.6 *Interface Viewpoint*

Users will require a front end interface to interact with the application and utilize its functionality. The design of these interfaces should focus on satisfying all user actions in a simple and appealing manner. Users should be able to intuitively navigate the application and perform essential tasks without confusion or error.

3.3.2.7 *Structure Viewpoint*

The design of this application describes the structure of independent modular components and their relationships to other entities. From a developer's standpoint, organizing the structure and interdependence of different components is essential when implementing a complex system. This application's design highlights key individual components along with their individual functionality and relationships with other entities.

3.3.2.8 *Interaction Viewpoint*

Each object component in the application should not need any logic to handle its interactions with other components. This functionality should be performed by services and model functions implemented in the back-end of the application. The design of this application aims to keep data and entities separate from the logical functionality required to extract and share data across different components.

3.4 System Architecture

3.4.1 Architectural Design

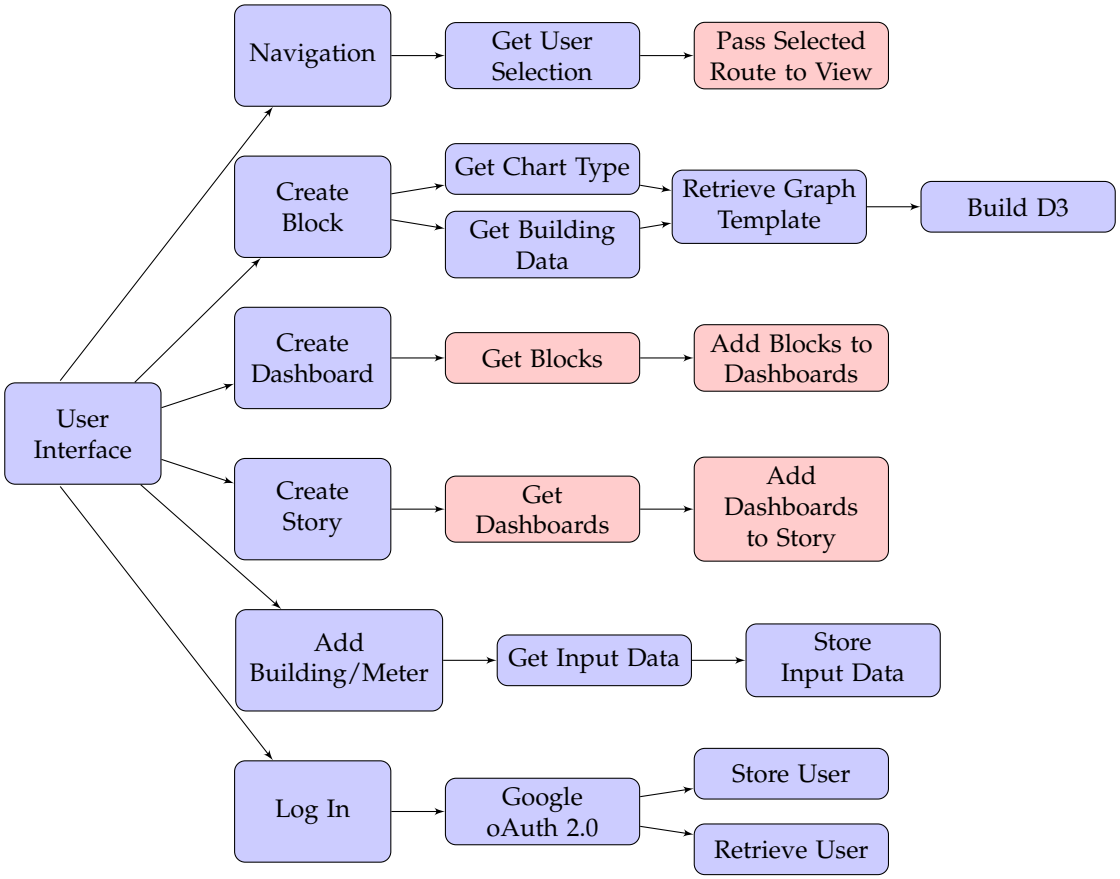


Figure 2: An overview of the major application systems.

3.4.2 Decomposition Description

All database operations (retrieving data and storing data) are denoted with red fill and will be carried out by an AngularJS "Service" within the component's controller.

3.4.2.1 Log In

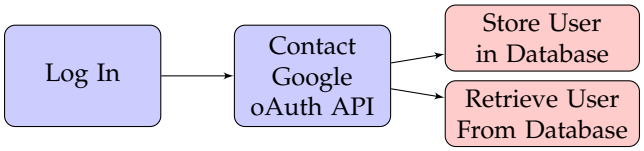


Figure 3: A functional representation of the log in subsystem.

3.4.2.2 *Navigation*

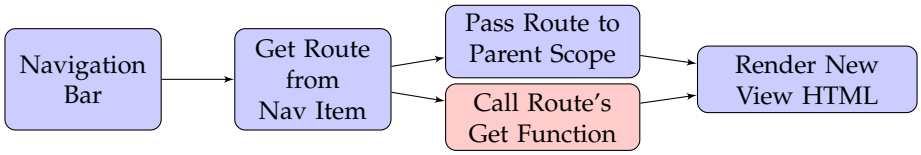


Figure 4: A functional representation of the navigation subsystem.

3.4.2.3 *Create Block*

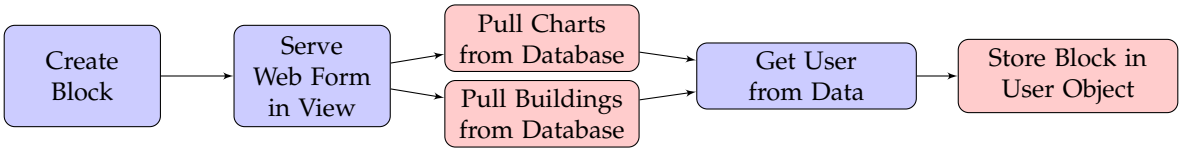


Figure 5: A functional representation of the block creation subsystem.

3.4.2.4 *View Blocks*

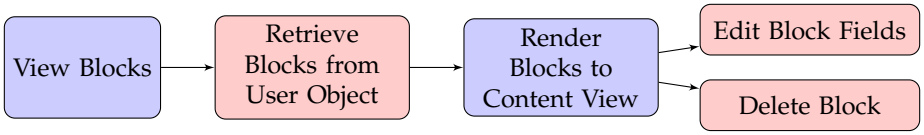


Figure 6: A functional representation of the block view subsystem.

User created blocks will be stored in their respective "Blocks" array This allows a user to keep track of the specific blocks they create and reuse them to create stories.

3.4.2.5 *Create Dashboard*

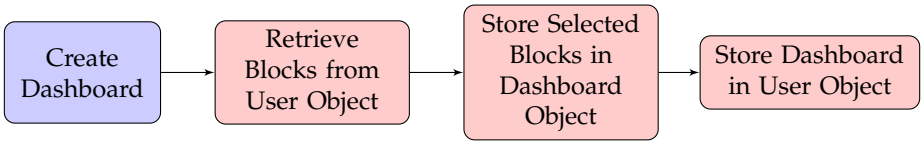


Figure 7: A functional representation of creating a dashboard object from user created block.

A dashboard is a collection of blocks. The dashboard component provides a way to group blocks together based on user preference and display them in the content view by simply rendering each block stored.

3.4.2.6 *Create Story*

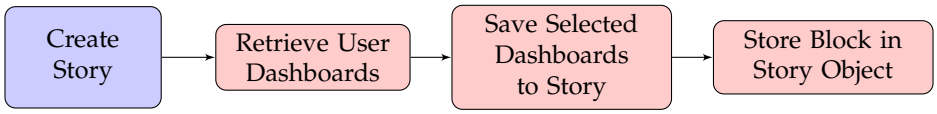


Figure 8: A functional representation of the story creation subsystem.

Stories are stored in the User object who created them in an array "Stories." This allows the application to render stories created by the session user easily and will eliminate the use of SQL style querying to retrieve stories based on a user key.

3.4.2.7 *Add Building/Meter*

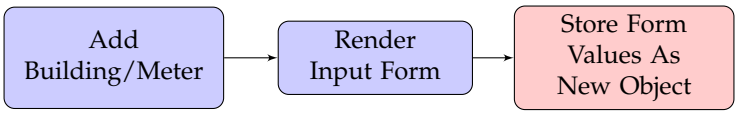


Figure 9: A functional representation for storing new building and meter objects to the database.

3.4.2.8 *Create Graph*

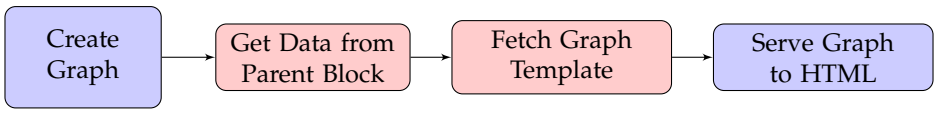


Figure 10: A functional representation for building a graph in a block.

To render a chart or graph to the webpage, our application will gather all the parameters from the block component that contains the graph (the building(s), the chart type, and the date range). The the graph component will then fetch a D3 template for its specific chart type, input the data from the block, and serve it to the dashboard.

3.4.2.9 *Get AcquiSuite Data*

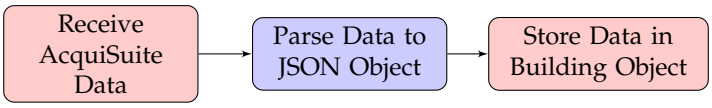


Figure 11: A functional representation for storing AcquiSuite data to a building object.

Timestamped data entries from AcquiSuite Servers will be contained in their respective building objects from which the data was sent. Similar to the rational behind storing blocks in the user object that created them, each building will have

an array of meter data sorted by timestamp. This will allow our application to use a service to retrieve the data for any graphs or charts directly from the buildings objects in the relative block. A graph service can then fetch the building data, apply filters (like date ranges), and pass it to a D3 graph to render. This minimizes the amount of total data to be parsed and collected by narrowing the data to the relative block.

[**Note:** Subject to change in later revisions. The methodology of how to receive data from AcquiSuite data acquisition servers is still being designed.]

3.4.3 Design Rationale

The system's architecture revolves around a heavily modular system where individual components have their own functionality and are able to work cohesively with other components. The modular architecture also breaks up the back-end components into smaller services which simplifies functionality across the entire application.

3.5 Component Design

This subsection is designated to describing individual components in greater detail, including the type of element, designated functionality, and the reasoning behind their design. Overall, this subsection aims to provide an in-depth view of the application components and describe their individual roles in the application as a whole.

3.5.1 Block

A Block is a component in our application that will act as a container for a graph, filter options, and provide information about the data in the graph.

3.5.1.1 Element

A Block is an HTML element that uses the Bootstrap 4 "card" component for its design. It will have a title bar with the Block's name, then a content container split into a grid. On the left, the Block will display the building(s) being graphed, contain a column of filter buttons, and display the D3 graph in the right column.

3.5.1.2 Functionality

The Block component neatly organizes data for the user by encapsulating the data, filters, and graph components into one container. A Block needs to pull data from the "building" objects that it contains, gather data from any filters, and pass all this data into the correct graph template to generate a D3 graph. There will be a service within the Block's AngularJS controller that will gather this data and pass the desired output to the dashboard.

3.5.1.3 Rationale

The Block component is an efficient way to create a reusable HTML template that can be reproduced throughout the application. This provides a simple and consistent process for displaying graphs and data. Additionally, the block component yields a compact solution for organizing data from multiple sources.

3.5.2 Dashboard

The dashboard component acts a way to store collections of block objects based on user preference.

3.5.2.1 Element

A dashboard is an object in the database that holds a collection of block objects.

3.5.2.2 Functionality

The dashboard object is a way to conveniently package blocks together into groups so users can reuse the same groups or create unique collections of different blocks. When "viewing" a dashboard, the application will simply iterate through the block objects and render them to the view container in a vertical list.

3.5.2.3 Rationale

The dashboard components provides a simple way to organize data keep track of user generated collections. The dashboard component will work nicely with AngularJS to render an HTML element for every object in an array, which in this case are the blocks in the dashboard.

3.5.3 Login

The Login component is an HTML button that will allow users to login with a Gmail account by redirecting the user to Google's oAuth 2.0 API.

3.5.3.1 Element

The Login function will be a button in the corner of the screen or somewhere in the navbar. The button will change to "logout" if the user is already logged in.

3.5.3.2 Functionality

Clicking the Login component will redirect the user to our authentication service. The application will use "PassportJS" as an authentication middleware which authenticates the user with Google oAuth 2.0 tokens and sets the req.user variable upon redirecting to the specified route. Once a user is logged in, they will have access to additional application functionality and navigation items based on role.

3.5.3.3 Rationale

From a developer standpoint, requiring users to login for certain features will make separation of privileges much easier. With authentication, users will also be able to have the convenience of Google SSO.

3.5.4 Navigation Bar

The Navigation bar is a commonly used UI feature that provides users the ability to navigate through different webpages of the application.

3.5.4.1 Element

The navigation bar will be an HTML element with differentiable “nav-items” that render different views to the content view space. There will be a fixed-top navigation bar for generic item and a side bar navigation for more explicit functionalities such as admin controls.

3.5.4.2 Functionality

Clicking on a nav-item has one of two functionalities:

- Redirects the user to a new webpage by having the route’s get function perform a redirect.
- Renders a new view to the webpage by having the route’s get function render a new HTML view.

The navigation bar shows which item is currently being served by changing the style of the active nav-item to be different than the rest.

3.5.4.3 Rationale

From a user perspective, having a navigation bar to navigate around the main features is an extremely useful UI design decision. From a design perspective, having a dedicated area to insert useful links is much more streamline than listing them. The navigation bar is a very useful tool for serving different content to the view space statically.

3.5.5 Graph Component

Our application will contain a graph component that accepts a graph type and data as input and constructs a graph using the D3 visualization framework.

3.5.5.1 Element

A graph component will be an SVG element, created with D3, that is served to the webpage inside of a block.

3.5.5.2 Functionality

The graph component’s functionality involves gathering different input parameters from the parent block component, fetching a template based on the graph type, and creating a unique graph. The graph component should have its own service to handle all the data organization and create a list of uniform parameters than can be passed to a D3 graph template.

3.5.5.3 Rationale

Having a structure like this simplifies the act of generating graphs by having a series of templates that can be fed data in a specific format. The graph service can take all the data and different filter parameters from the block object and format it into the right structure. All the computationally heavy functionality is black-boxed in the graph service and keeps the functionality handled by the user and front-end at a minimal. Another benefit provided by the graph component is that it can change the graph dynamically. Our application may include filters in the block component like a date range or a filter to change the graphed variable, which can trigger the graph service to reconstruct the data and create a new graph to serve based on the applied filters.

3.5.6 Story

A story is a container for user dashboards to be stored and retrieved. A story is an abstract object that allows the user to create unique collections of dashboards with general theme or purpose.

3.5.6.1 Element

A story will be an database object that contains collections of dashboards.

3.5.6.2 Functionality

Each story will have a collections of dashboards. The number of dashboards in the collection will vary from story to story based on user preference.

3.5.7 Content View

The content view is a portion of the webpage where different views can be rendered to provide different content.

3.5.7.1 Element

The content view will be an HTML container that resides within the context of the webpage. The user will be able to change the content of the container by selecting different navigation items.

3.5.7.2 Functionality

When the user clicks on a button or list on the side bar, it will pass a route to the scope variable in the view container. Once that route is called, the Angular function “ng-include” will inject the rendered HTML into the context of the webpage.

3.5.7.3 Rationale

From a developer standpoint, having a content view will dramatically reduce clutter both in the code and in the UI. The content view allows the application to serve dynamic content without reloading the webpage and creates a simple way to perform navigation services.

3.6 Human Interface Design

3.6.1 Overview of User Interface

3.6.1.1 Public User

A public user is a user without a registered account or a user that is not logged in to their registered account. The authentication of a user is based on if they have a registered Oregon State Gmail domain. Public users will have limited access to features and data on the website. For example, the left side navigation bar will be hidden from a public user. Public users will not be able to create personalized dashboards or stories. Public users will be allowed to view public user generated dashboards as well as general building dashboards. Essentially, public users will not be able to create personalized data sets or modify elements of the site but will be allowed to view general public data.

3.6.1.2 Authorized User

An authorized user is a user with a registered Oregon State Gmail account but does not have administrative privileges. When an authorized user is logged into their account, they have greater privileges than a public user as well as all privileges granted to a public user. They are allowed access to a left side navigation bar that has special functionality. Unique functionality granted by the vertical navigation bar include: viewing of personal and public dashboards, viewing of personal and public stories, and access to personal settings. Authorized users may create personalized dashboards and stories for unique purposes.

3.6.1.3 Administrative User

Administrative users will have the highest privileges and control of the website. They have all privileges as a public or authorized user. Administrative users will be able to add or remove buildings, meters, public dashboards, and public stories as well as perform general administration tasks.

3.6.2 Screen Images

3.6.2.1 Home Webpage (Public Access)

link: See Appendix B for image

If a general public user does not have an account but wishes to access energy monitoring data, they will have a unique experience with the site that is different than those who are logged in to authorized accounts. For instance, they will not have the ability to create personal dashboards or stories, but will be allowed to view public stories, public dashboards, and general building dashboards. They will not have access to the same navigation bar on the left side of the screen as a logged in user.

3.6.2.2 Home Webpage (Logged in Access)

link: See Appendix B for image

If an authorized user wishes to access energy monitoring data, they will be able to have special privileges. For instance, they will have the ability to create personal dashboards or stories which they may specify as private or public. They will have access to a navigation bar on the left side of the screen which allows for unique data viewing and privileges.

3.6.2.3 Buildings Webpage

link: See Appendix B for image

Users may select the “Buildings” tab in the navigation bar to bring up an array of building cards to choose from. Every building on campus that is gathering metering data associated with this application will have a card in this array. A user may select a building card to bring up a generalized dashboard of information for each specific building. The user will be able to search for buildings by name in a search bar. Administrators will be able to add buildings as the metering system expands to new buildings.

3.6.2.4 Selected Building Page

link: See Appendix B for image

Once a user has selected a building card, a generalized dashboard will be displayed to them. All buildings associated with this application will contain the same generalized dashboard to be viewed. These dashboards will display various cards of data detailing the most general information about their respective buildings. A user may full screen the application to remove menu clutter and allow for a more presentable display of the data. Users will have the option to modify personal dashboards or public dashboards by adding a card, adjusting the organization of cards, or editing the information on the cards.

3.6.2.5 Dashboards Page

link: See Appendix B for image

Users may select the "Dashboards" tab in the navigation bar to bring up a list of user-generated dashboards. The dashboards will contain user specified arrangements of data cards. The list of dashboards will detail the name of each dashboard, the user that created each dashboard, and the date and time of when each was created. The user will be able to search for dashboards by name in a search bar. A user may select a dashboard from the list to view or use the options menu for more specific tasks.

3.6.2.6 Selected Dashboard Webpage

link: See Appendix B for image

Once a user has selected a dashboard, a user-generated dashboard will be displayed to them. The dashboard will display various cards of data presenting information about various buildings energy usage. A user may full screen the application to remove menu clutter and allow for a more presentable display of the data.

3.6.2.7 Story Webage

link: See Appendix B for image

Users may select the "My Stories" tab in the navigation bar to bring up a drop down list of their personal stories in the left side navigation bar. Each story will contain a list of dashboards that was generated by the current user to their preference. The list of stories will include both their private and public stories. A user may select a story from the list to view or use the options menu for more specific tasks. include the original document, showing what you thought, at the time, was the project design Add (your client should have okay'd): What design aspects were changed, deleted or added? Why?

4 TECH REVIEW

4.1 Daniel Technology Review Overview

4.1.1 Introduction

My role in this project will be integrating the front-end with the back-end framework and acting as a dev ops engineer. I want to take on the role of dev ops for this project to hone in on certain skills that have not previously been targeted in my past web-development internships. I have created UX/UI and built pages/forms that manipulate database data. I have written back-end model functions to retrieve data for use in an application. Now I want to manage the full-fledged construction of a full stack web application. Another motive for taking a dev ops role is with a framework like NodeJS, there are endless amounts of libraries and frameworks that you can utilize in your application that can simplify every component. What I want to take away from this course is a well-rounded knowledge about how different components of an application work together and how to integrate each part efficiently.
By taking a dev ops role in this project, I will be able to:

- Transition away from the tasks I've been previously trained on.
- Help my teammates understand things that I have previously learned by not implementing them myself.
- Develop a more comprehensive understanding of NodeJS, Angular, Express, and NOSQL Databases.
- Take lead in developing a high-end development environment for my team and I in order to create efficient workflow.
- Implement components of our web application that are new to me.

There are a lot of different components that go into making a web application work and it takes a high-level understanding to implement them all. Our web application will act like an admin dashboard that displays specific charts and graphs based on user preference.
Firstly, from a dev ops perspective, we need an architecture that can host a NodeJS application from a single VM. We will be using Amazon Elastic Compute Cloud (EC2) to host our web server. This EC2 instance will listen for inbound requests and transfer them to our Node server daemon. Our Node server will then have to communicate with our MongoDB to retrieve the necessary data.
Secondly, we will need a workflow architecture for continuous deployment that can be utilized by all members of the team. We will use Github to create a "Git-flow" environment that proceeds as follows:

- The repository master branch is considered to be "production."
- Developers will create "feature branches" branched from master.
- Feature branches will be pull requested, reviewed, and merged into the master branch.

Git-flow allows team members to work on separate features while building off of the current working application. Feature branches make sure that the master branch is always in working condition and no new code gets merged without review and confirmation that it works.
Thirdly, our web application also has users with different roles, which adds a lot of overhead to the application. Our application needs to have user authentication, user sessions, user roles, and code to enforce it all. Using Google's oAuth 2.0 authentication, our application will force users to log in with their Google accounts and Google will handle most of the overhead. When a user logs in, "the Google Authorization Server sends [the] application an access token" which is then stored in the database instead of a user password [12]. Then, through utilizing the Passport.js authentication middleware, we can create user sessions and manage authentication with this Google token and not have any security issues with passwords.
The final main component of our web application is the actual functionality of providing data visualizations to the user in a clean and dynamic way. This will include retrieving and parsing data from Acquisuite data acquisition servers, storing this information in the database, and creating dynamic graphs and charts using a visualization framework that can be manipulated by the user. This will involve developing filters and date ranges that can be applied to all the data to select sub-portions of data based on specific buildings, collections of certain buildings, or time periods. Having a visualization framework that works well with AngularJS and our MongoDB database is essential in creating a smooth, dynamic web application.

4.1.2 Visualization Frameworks

Our web application will provide near-real time data visualizations for energy consumption by buildings on campus. This application will need to dynamically create charts and graphs based on energy data from the database. A key to choosing a visualization library will be to find one that can be dynamically created and changed as new data is received from the data acquisition servers, and the ability to create chart templates that can be reused on multiple pages with different input parameters.

Basic Criteria

- Handle dynamic data.
- Allow the user to manipulate data by applying filters or selecting data points.
- Works well with Angular framework.
- Templatable.

4.1.2.1 D3.js

Repository Commits: 4,104
Contributors: 120
D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document [5]. It is extremely well documented and widely used. D3 has a high learning curve for beginners to take head-on, but has a wide array of different visualizations and customization.

Pros

- A lightweight, versatile javascript library that uses "SVG to create the graphical elements" and appends them to DOM elements [15].
- Makes use of javascript functions and DOM controlling functionality to dynamically change the content of the page.
- Provides a lot of variety and ability to customize graphics.
- Widely used and there is a lot of documentation and resources available to assist the learning and development processes.

Cons

- D3 is essentially an API to to manipulate SVG, it is not a charting library in of itself [16].
- You cannot easily pass a data set into a specified chart type like other libraries.
- A large amount of "up-front investment in time to get a handle on the D3 language" [17].
- Angular and D3 both attempt to control the DOM and so you have to find a way to make the two work together which is counter intuitive to both framework's APIs.

4.1.2.2 Vis.js

Repository Commits: 3,165
Contributors: 137
Vis.js is a lightweight charting library that allows users to create clean charts from dynamic data sets [6]. It is responsive and allows for interaction with the data on the page. Vis.js is praised for it's network chart capabilities but is limited in the number of different modules you can create.

Pros

- Easy to use and less of a learning curve than D3.
- Allows for interaction and manipulation of data on the chart.
- Able to handle large amounts of dynamic data.
- Really clean and nice looking graphics.

Cons

- Limited amount of possible chart types.
- Does not have built in heat map.

4.1.2.3 Chart.js

Repository Commits: 2,465
Contributors: 236
Chart js is a very lightweight library that provides 6 chart types and fully responsive designs. ChartJS is well documented and easy to use, but lacks in variety.
Pros

- Uses HTML5 canvas element.
- Allows for easy creating based on chart type specification.
- Library provides Line Charts, Bar Charts, Radar Charts, Pie Charts, Polar Area Charts, and Doughnut Charts.
- Very responsive charts based on screen width.
- Simple API, easy to use.

Cons

- Limited amount of possible chart types.
- Does not have built in heat map.

4.1.2.4 Conclusion

In conclusion, despite the steep learning curve associated with D3.js we think it will be the best option for our web application. It has the widest range of available graphs to accommodate all the client's requirements and desired visualizations. There are also a number of wrapper libraries available for D3.js like DC.js and dimple.js to help create charts from D3. This is a great way to get around the inconveniences and downsides to D3.js and reap the benefits of all the other charting libraries. Another benefit to using D3 is the extensive amount of templates, examples, and documentation that exists to help guide the process and implementation of our application.
It is possible that our decision will change as we begin creating the web application and find other libraries to fit our needs. There are a lot of AngularJS libraries that specify unique directives for implementing different charting libraries and we might find one throughout the midst of our development that handles all of our needs swiftly and effectively. One of the main factors that guided our decision towards D3 was the ability to create heatmaps, which was a client requirement. If this requirement falls through, or we simply use D3 only for heatmaps, we may be able to implement a more lightweight charting library to achieve better simplicity.

4.1.3 Means of Incorporating Authentication

Our web application will have an authentication layer which will allow users to register for our application and design their own "stories." Our application will also authenticate user roles so that administrative users will have access to exclusive parts of the application. We want a simple way of authenticating users, while keeping personal information safe.
Basic Criteria

- Remember users across web pages (sessions).
- Allow users to be added to the database.
- Protect sensitive information like passwords.

4.1.3.1 Building Our Own Authentication Layer

Node.js has a lot of helpful modules and packages that allow you to create your own password hashing functions and generate a custom authentication layer. There is a "crypto" module that is included in Node.js that "provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions" [18]. In addition to the crypto module, there are numerous Node.js extension modules that perform different hashing functions and provide the same functionality as the native crypto module. There are modules like *bcrypt* and *scrypt* that use different hashing algorithms to salt and hash password into a fixed length string to be stored into the database.

link: See Appendix A for code listing

Pros

- Creating our own authentication layer would provide us full control over how passwords are hashed and stored into the database.
- Provide understanding of every component that goes into our application's authentication.
- Do not have to rely on another API to authenticate users.

Cons

- Laborious work and very time consuming.
- A lot of room for error and the possibility of data being compromised.

4.1.3.2 Outsourcing Authentication to Google

Google oAuth 2.0 is a Google API that authenticates users by signing in with their google accounts. There is a lot of documentation about how to integrate Google's authentication API with Node.js. *Passport* is authentication middleware for Node.js that provides simple authentication using "A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter," and Google [19].

link: See Appendix A for code listing

Pros

- Takes all the security risks out of creating our own authentication.
- Saves a lot of development time.
- Widely used with a lot of documentation.
- Relieves the need to store user passwords in the database.
- Can use the passport authentication middleware to simplify authentication even further.

Cons

- Limits users to having a google account.
- Relies on Google API to be working and running.
- Adds dependencies to the project.

4.1.3.3 Use CAS (Central Authentication Service)

CAS is an authentication process that redirects users to a CAS login page like the one that Oregon State University uses for most things like Canvas, Box, and MyDegrees.

CAS uses sessions and a CAS Client server to authenticate users throughout a web application: *When CAS redirects the authenticated user back to your application, it will append a {ticket} parameter to your url.* The ticket returned to your application is opaque, meaning that it includes no useful information to anyone other than the CAS Server. The only thing that your application can do is send this ticket back to CAS for validation. CAS will then either respond that this ticket does not represent a valid user for this service, or will acknowledge that this ticket proves authentication. In the later case, CAS will also supply the user's NetID so that you know the identity of the user [20].

Pros

- Can use the OSU official CAS to provide a nice theme.
- Keeps the same centralized CAS server as the rest of Oregon State University's application.

Cons

- Relies on Oregon State University's CAS server to be operational.
- Adds extraneous code and implementation.
- Not as modular as the Node.js/Passport implementations of oAuth 2.0.

4.1.3.4 Conclusion

After reviewing the different means of including authentication for our web application, we think that using the Passport middleware to implement Google's oAuth 2.0 API would be the best option. It can be easily integrated into Node.js applications, everyone at Oregon State University has a Gmail account, and it removes the need to store any passwords into our database. The Google oAuth 2.0 authentication process uses a token that is received from the Google API that we will store along with the user ID and name into our database.
Another large benefit to using the Passport.js and Google oAuth 2.0 authentication process is that passport handles most of the overhead involved with user sessions and per-page authentication. Using a small cookie set that contains the user id, "Passport will serialize and deserialize user instances to and from the session" to handle login sessions throughout the application [21].

link: See Appendix A for code listing

Using Google authentication and the Passport middleware will drastically simplify the authentication process of our application and keep the amount of sensitive data in our database to a minimum.

4.1.4 Front-end Framework

Our web application will be a series of dashboards to display energy data based on different buildings and subsets of buildings on Oregon State University's campus. A key part to designing a clean dashboard is having a well-spaced grid-like layout with different charts and graphs to display a multitude of different data sets and trends. Rather than customizing classic html elements and using div containers to space our dashboard, we wanted to look into bootstrapped dashboard templates that allow easy customization and clean-looking results.
Basic Criteria

- Integrates well with Angular and the entire MEAN stack.
- Components can be easily integrated with Visualization Framework.
- Customizeable.

4.1.4.1 CSS Bootstrap

Repository Commits: 17,255
Contributors: 953
CSS Bootstrap is the most widely used CSS framework available. It has the most expansive component list and helps developers create clean and beautiful UX/UI in minimal time frames. Bootstrap allows developers to get their publications up and running quickly without having to worry about front-end styling. A major issue with Bootstrap is that it tends to look very similar across applications and leaves little room for simple customization.
link: See Appendix A for code listing

Pros

- Extensive component list, responsive design, and built-in Javascript functions. [22]
- EFully responsive design.
- Huge developer contribution and maintenance.
- Used by major companies like Lyft.com, Vogue.com, Vevo.com, and Newsweek.com. [22]

Cons

- Unsuitable for small scale projects. [22]
- Not good if you want to have large control over UI.

4.1.4.2 Pure CSS

Repository Commits: 541
Contributors: 51
Pure CSS is known for its lightness and simplicity. Because it's small, it is very fast loading and makes for a lightweight web application. It is also unique in that it allows modules/components to be downloaded individually, reducing its size even more. Pure CSS is good for small projects that need to get up and running quickly and easily.

Pros

- Meant for small project to get up and running quickly.
- Responsive design by default.
- Pure CSS is modular so you can download only the components you need.
- The complete module is very small so it is quick loading.
- Able to be used complimentary with other frameworks.

Cons

- Not as extensive component list as Bootstrap.

4.1.4.3 Foundation

Repository Commits: 15,094
Contributors: 959
Framework is the second most popular css framework on the market behind Bootstrap and tends to perform just as well. Unlike bootstrap's very prominent theme, Foundation allows for more customization with the look and feel of web pages [23]. Foundation also has a very good grid system to make the layout of components clean and responsive [24].

Pros

- Responsive design by default.
- Easier to customize than Bootstrap.
- CSS classes are built in. [25]
- More unique look than the more-popular Bootstrap.
- Good grid implementations with customizable grid layouts.

Cons

- Less maintained than Bootstrap.
- Lack of support.
- Higher learning curve than Bootstrap.

4.1.4.4 Conclusion

A lot of online resources acknowledged the fact that it is hard to make a website not look like Bootstrap when using Bootstrap css. Despite this, we are choosing to use Bootstrap as the main front-end framework for our application. Similar to our reasoning behind choosing D3.js, we would like to use a framework that is heavily supported and well-documented, as it will be easier to answer questions during development. We think that the time spent restyling Bootstrap to make it look unique will not compare to the time saved by utilizing a well-documented framework. There is also a project called UI Bootstrap that works with AngularJS to create directives for each of the Bootstrap components [26].

4.2 Parker Technology Review Overview

4.2.1 Introduction

Our scalable web application for monitoring energy usage on campus is an application that utilizes various technologies to accurately store, process, and display energy usage of buildings on the Oregon State campus. There are many different components and frameworks by which to build an application for this purpose and selection the right technologies for is proper functionality is crucial. Choosing technologies that may be outdated, have flaws specific to the needs of the application, contain compatibility issues, etc., could prove to be detrimental to the project as a whole. It is necessary to analyse the various technologies and select the technology that will provide the best performance and functionality prior to the creation of the application so that the development process will be fluid and efficient.

This document will discuss the pros and cons of various specific technologies that may be applicable to the components of our web application. Specifically, this document will compare technologies for the database framework of the application, the database host of the application, and the server framework that will be used. For the database framework, comparisons will be made between mongoDB, mySQL, and Apache Cassandra. As for our database host, comparisons will be made between AWS EC2, Microsoft Azure Cosmos DB, and Google CE. And finally, for our back-end framework, comparisons will be made between Node.js, PHP, and Golang.

4.2.2 Technology 1: Database Framework

For any web application that needs to utilize compilations of data for its processes it is critical to have a reliable and powerful framework for that data. Specifically, for our application we need to select a database that can handle a large amount of raw text based data that is presented to us from our energy metering systems. Having a database with proper functionality and compatibility with the rest of our technologies is necessary so that our application can request and store data in meaningful ways.

4.2.2.1 mongoDB

MongoDB is a noSQL scalable open source JavaScript based database system. It is a non-relational database and it's documents are stored in json format essentially making all JavaScript data types compatible with this database. It is often used in conjunction with Node.js projects because of this relationship with JavaScript. It is widely used because it is a "document" based database which essentially allows the data that it stores to be interpreted as objects of data, where you can have levels and sub-levels of data in one data entry. For example, you could have 1 entry in mongoDB that represents a store, and within that store object it may have lower levels of data such as products provided that store or employees that work there. It is widely used as the database for many web based applications because it is designed to have useful models for data structures such as logs, graphs, account and user profiles, form data, etc. It has been used by many major industries and companies such as MetLife, government websites, Expedia, and more. MongoDB is often referred to as new user friendly because it is widely documented and has many tutorials online.

4.2.2.2 mySQL

MySQL is perhaps one of the most popular open-source relational database management systems, written in c and c++. It is an extremely well documented and mature database system and is used widely by many applications, include web based applications. It is compatible with nearly all operating systems and supports many storage engines. Data using mySQL is stored in tables that may be accessed or manipulated using query requests. It is not a document oriented database system so data may not have various levels of hierarchy within each entry, but each entry may be logically bound to other parts of the database which would be managed by those who are designing the database. MySQL is proven reliable, high performance, and easy to use and is used by major company applications such as Facebook, Twitter, YouTube, and much more.

4.2.2.3 Apache Cassandra

Apache Cassandra is an open source linearly scalable storage system that is build to handle large amounts of data. It is build using two core technologies: Google Big Table and Amazon Dynamo. It is a noSQL class of database system developed to improve consistency, availability, and partition tolerance of the database. It is designed to handle arbitrary data types that may have interconnected relationships to other pieces of data within that database. Because of the way that Cassandra's architecture is designed, it is incapable of having a single point of failure among its nodes. Each node within the database is connected and running concurrently, there is no "master-slave" relationship between nodes and nodes can be added to an existing cluster without taking down the server. This works well for applications that need to have access to a database at all times. Cassandra excels for real world uses such as messaging, fraud detection, recommendation engines, playlists or collections, and more. An issue that presents itself is that it is not as widely used as either mongoDB or mySQL, and as a consequence is lacking the resources to learn about it as efficiently as either mongoDB or mySQL.

4.2.2.4 Conclusion

Our choice for the database framework for our application is mongoDB. Not only is it non-relational, provides us with scalability, and provides adequate functionality, it is a JavaScript based framework that is very compatible with the rest of our applications technologies. It is widely popular, relatively easy to learn, and allows us to access and store data in ways that will suite our needs for this assignment. While the other database framework technologies that were listed are potentially valid options, mySQL is not non-relational and they both are not JavaScript based technologies that fall under the "MEAN stack" development method, which is ultimately what our project will be utilizing.

4.2.3 Technology 2: Database Host

Web based applications and their associated databases are required to be hosted from some machine that is handling all of the processes that are required by the application for it to function and provide dynamic usability. A modern and reliable way to host a website and its database is from cloud services that will host your site with varying criteria based on various payment options or plans. We prefer to host our application on a cloud based platform so as to reduce any complications that may arise from hosting our site on a local machine that may not be nearly as reliable, secure, or structurally dynamic as cloud based services. Specifically, we will compare and contrast the cloud service providers AWS EC2, Microsoft Azure Cosmos DB, and Google CE and decide which is best to suite our needs.

4.2.3.1 AWS EC2

Amazon Web Services (AWS) provides a secure, scalable, and non-constricting web based service for applications which may use the non-relational database that we chose mongoDB. It allows complete control of the application (just as root access to computing capabilities) and scales elastically to the needs of the application. Along with control and scalability, the operating system environment in which the application is to be ran on may also be chosen. It is reliably available, secure, and has access to many other AWS computing applications that may be utilized by the project, offering the widest range of services to its users. It is designed to be simple to learn and cost effective. Because of AWS many features, it has been an excellent choice for a huge number of applications to be ran on, including Netflix, AirBnB, and Lamborghini. Companies have opted to use AWS because of its many great features, specifically cost of use, freedom of control, reliability, and ease to learn.

4.2.3.2 Microsoft Azure Cosmos DB

Microsoft Azure Cosmos DB accomplishes a similar functionality as AWS EC2 but is not nearly as widely used as AWS. It has support for applications based in non-relational database services such as mongoDB. Essentially it offers all the functionality of a privately owned server but based in the "cloud", similar to AWS. It is open source and scalable, as well as provides security, good pay rates, and access to many of Microsoft Azures cloud computing applications. It has low latency and fast I/O performance, which may be scaled up for a larger price. Notable users that rely on this service

include Pearson, Ford, and NBC News. In a lot of ways it is very similar to the other two technologies compared here, but Azure is perhaps less reliable than AWS and Google CE, having had a series of outages over the years.

4.2.3.3 Google CE

Google Compute Engine (CE) is Google cloud computing equivalent to Microsoft Azure and AWS. It is similar to other cloud computing services in that it supports all of the quality attributes of a good web hosting service, including scalability, security, control, area of distribution, reliability, support for non-relational databases such as mongoDB, etc., but it is not nearly as widely used or distributed as AWS. It may have an advantage over the other services with regards to machine-learning capability that an application may take advantage of, but in our case we do not require that unique resource. Google CE may be more appealing to niche markets, as it focuses on smaller scale companies. Essentially, Google CE is very similar to AWS but is much less widely adopted and does not offer as many resources to use as AWS EC2.

4.2.3.4 Conclusion

The database hosting technology that we have opted for is the Amazon based service "AWS EC2". The main reasoning for this decision is that it has a massive selection of other applications that may be utilized for our project, more so than either of the other two options. As well as more provided services, AWS EC2 is without question the dominant hosting service in the market because of its reliability, security, and scalability. It offers exceptionable payment plans and is widely used. Other database hosting services may work fine for our application, but AWS EC2 provides us with more potential services and is more widely trusted by large scale companies.

4.2.4 Technology 3: Back-End Framework

A solid back-end framework is a necessity for any scalable application that requires dynamic and fluid processing and presentation of information. While a solid database is necessary, it is useless without a proper tool for accessing and manipulating that data efficiently. Our web application will require present dynamically changing informative entities within the page that will need to neatly and quickly display our data in formats that we desire, so a proper back-end framework technology will be crucial to accomplish this. Comparisons will be made between node.js, PHP, and Golang to be used for our application.

4.2.4.1 Node.js

Node.js is an open source cross-platform runtime environment for developing applications using JavaScript. It can be run on various operating systems such as Linux, Windows, or OS X. It has a massive open source library of JavaScript libraries that may be easily installed and used by an application. It is extremely fast, scalable, asynchronous and event driven. Essentially, this framework can make multiple calls to a server without needing to be hung up on one specific call, it can reference a database dynamically. It is very popular and widely used by many large companies such as eBay, General Electric, PayPal, Yahoo!, and more. It performs exceptionally for applications that need to produce data functions quickly and dynamically so as not to refresh a browser to display data.

4.2.4.2 PHP

PHP is a widely used and matured general purpose scripting language. It is perhaps more widely used than JavaScript as it is an older developed scripting language. PHP is perhaps simpler than node.js as it does not require a server, simply a php file that may be interpreted by a web server. PHP uses a multi-threaded blocking I/O to achieve parallel task completion whereas node.js uses a single threaded non-blocking I/O model. Many large companies also use PHP for their products, including Facebook, Wikipedia, Flickr, and many more big names. PHP is reliable and simple to use, but lacks modern functionality, organization, and resources that node.js has. As stated before, our application is JavaScript dependant so it is wise to keep as many technologies as possible within this language.

4.2.4.3 Golang

Golang, commonly referred to as "Go", is a more modern back-end framework developed by Google. It is not as widely used as node.js, but it is still and effective choice for many web based applications. Because it is in its early days of development use, it is still a relatively immature framework, but that is not necessarily a bad thing. It outperforms node.js with speed and performance of tasks because it is not based on an interpreted high language such as JavaScript. It is used by companies such as Google, Docker, and Dropbox because of its excellence in scalability and concurrency. Large scale data can be effectively managed using Go but it may lack the broad scope of tools that node.js could provide. A major con of utilizing Golang is because of its relative youth. It lacks the wide range of tools that we may utilize for our application, it lacks the simplicity of node.js, and it is not a JavaScript based environment such as node.js.

4.2.4.4 Conclusion

Our application is very dependant on our back-end framework being able to dynamically and concurrently manipulate data on our web pages. While PHP and Golang are capable of doing this, they do not do it as simply as node.js. All of the technologies that are compared here are capable of accomplishing the simpler tasks that our application needs, such as registering user data and doing simple read and writes to our database, but node.js provides a more simplified way to accomplish the more complex tasks and is compatible with the rest of our technologies. Having a JavaScript based back-end framework is also a wise choice as a large portion of our technologies used for our application are also JavaScript, reducing complexity of component interaction.

4.2.5 Conclusion

We have made a decision to base our application off of the JavaScript language, and as such we have chosen many of our technologies based on this so that they may be compatible and perform without complications. Reducing the number of languages involved in a full stack application reduces the complexity of the project and allows development to happen more fluidly. We have opted for JavaScript based technologies because of it's wide use, large amount of applications and libraries to use, and overall high-level simplicity. Many of our technologies used, specifically our back-end framework and database framework, will utilize JavaScript based technologies, such as mongoDB and Node.js. Not only do these technologies have sufficient performance, scalability, and scope for our project, it allows the project to be easily compatible within itself. As a matter of our database host, all we need is a sufficient web service that can provide support for a non-relational database system such as mongoDB. The best choice to us is using AWS EC2, which supports non-relational database systems and allows us a vast array of tools provided by Amazon Web Services.

4.3 Aubrey Technology Review Overview

4.3.1 Introduction

My role in this project will primarily focus on the back-end side of the web application. This means I will oversee ensuring the database does exactly what it should be doing for any given query. This also includes any security measures to protect the database from common malicious attacks. Another thing I will oversee handling JSON requests from the front-end. This means I will have to ensure that the data expected by the user is returned by the server. Finally, a task I will oversee is to ensure the general structure of the server code as well as ensure computational logic is computed correctly.

Since our application for measuring energy usage on campus is web based; we will need various tools, technologies, and frameworks to assist us in storing and displaying information correctly. For this reason, it will be of the utmost importance to ensure that the right technology is selected for our project. We want to avoid technologies that are outdated, flawed, or otherwise counterintuitive for our needs. It is thus necessary to analyze multiple different technologies for each category to guarantee we are using the best technology available.

This document will discuss both the pros and cons of different technologies that will be required for various components of our application. More specifically, this document will compare technologies for three different components needed by our application. These components are: Structural Framework, Server-side Web Application Frameworks, and Web

Hosting. For the Structural Framework, comparisons will be made between Angular.js, Backbone.js, and Ember.js. For the Server-side Web Application Framework, comparisons will be made between Express.js, Koa.js, and hapi.js. For the Web Hosting, comparisons will be made between Amazon's AWS, Oregon State University's Drupal through CWS, and Heroku.

4.3.2 Structural Frameworks

Structural frameworks control how the project directory and all related files/subdirectories will be stored. They aim to automate most of the overhead that will occur with most web based applications. Each framework will usually provide built in libraries to access databases, templating frameworks, and session management. Structural frameworks will run client side.

4.3.2.1 AngularJS

Pros

- AngularJS is extremely fast and easy to set up and run on server. By simply adding a few attributes to the HTML tags, it will perform all the heavy lifting to identify controller functions [27].
- It will then create dependencies between templates and the controller, which will then deliver the data directly to the screen [28].
- AngularJS is also extremely comprehensive. It provides a completely seamless option for front-end development.
- It does not require any other framework or plugin [27].
- It also features REST actions, which provides nearly instantaneous communication between clients and the server [28]. The REST actions also assist with retrieving interactable data within the webpage.
- It also features markup for DOM, which increases simplicity, integration with existing apps, and extensibility [28].
- Finally, AngularJS boasts a high performance. From all the features AngularJS has, it still yields extremely high speeds and fast response times [27].

Cons

- AngularJS mandates support of Javascript. Some computers and networks disable JavaScript by default. This means that the user would be unable to load, see, or access our page.
- AngularJS also relies heavily on the Model-View-Controller architectural patterns [27]. This means that we might have more difficulties since we aren't completely familiar with it.
- Since AngularJS is also quite layered and hierarchically arranged, it also might be difficult to debug scopes.

4.3.2.2 Backbone.js

Pros

- Backbone.js is a highly compact and extremely versatile.
- It mainly excels in smaller projects. This is because Backbone.js itself is a very minimalistic [29]. It is often referenced as a Javascript Library rather than an entire framework because of how small it is [29].
- It only contains the bare minimums needed to structure a web app in the Model-View-Controller pattern.

Cons

- Because of the barebones nature of Backbone.js, any non-basic features will have to be pulled in from separate libraries or plugins [29]. This will require a chunk of time for researching compatible libraries.
- Backbone.js also doesn't support two-way binding, which will often result in a lot of boilerplate for updating the model [29].

4.3.2.3 Ember.js

Pros

- Out of the three options, Ember is by far the most robust of the frameworks. It comes with a plethora of features.
- By itself, it will have everything needed to construct a web application, with some exceptions [29].
- This obviously would allow us to not have to worry about external libraries, dependencies, or plug-ins to get the application running [29].

Cons

- However, the high level of control comes at a cost. Since it's such a robust framework, there always might be a concern for performance/speed issues.
- Ember also has issues with adding any extra dependencies without breaking code [29].
- Ember is notorious for having extremely outdated examples and documentation [29]. Its official tutorials are also out of date, which raises the concern of where to find the answer when a particular issue comes up.

4.3.3 Server-side Web Application Framework

Server-side web application frameworks are, as the name implies, server side. This means everything in the framework will be executed server side, with minimal to no toll on the user's local performance. These frameworks must function in a stateless fashion. These frameworks will allow server side changes without having to refresh the page.

4.3.3.1 Express.js

Pros

- Express.js excels at being both lightweight and flexible. It is simple option for single-page, multi-page, and hybrid web applications [30].
- It is a powerful tool, despite being a micro-framework.
- Due to its simplicity, testing and quality assurance are streamlined.
- Its API takes advantage of the Node.js package manager to help install and distribute any needed plug-ins [30].

Cons

- Express.js doesn't have any recommended method of organization, which can lead to an unorganized codebase [30]. This can easily lead to non-reparability for some projects.
- Express.js is extremely manual labor-intensive when it comes to tasks. This can affect memory usage and performance if not addressed and handled professionally.

4.3.3.2 Koa.js

Pros

- Koa.js is very similar to Express.js. In fact, it was developed by the same team who created Express.js [31]. The developers created Koa.js to be a more concise and vivid alternative to Express.js.
- Koa.js is focused primarily on creating web applications and API's with improved and optimized performance.
- The main advantage it has over Express.js is its usage of the ES6 Generator feature [31]. This generator avoids callbacks all together, which increases the manageability of the code. It also assists in error handling and block building.

Cons

- The documentation for Koa.js is fairly lacking. This lack of documentation could prove problematic if we were to run into an obscure issue with the framework.
- Koa.js is also said to be an unstable framework [31]. Instability is something we cannot afford to have with our project since we will have a lot of data going at any point.

4.3.3.3 Hapi.js

Pros

- Hapi.js is a great choice because it comes with built-in support for input validation, caching, and authentication.
- There is an extensive list of third-party plugins available compatible with Hapi.js that will provide Object Relation Mapper and Object Document Mapper support [32].
- Object Document Mapper support will be important to our NoSQL themed database.
- Hapi.js also grants an extensive amount of control over request handling. It is also a plugin-based architecture that would allow us to pick and choose which modules we want [32]. This would give us more control over how large the application is scaled.

Cons

- Hapi.js doesn't come with a built-in code structure, which would require us to build our own [32].
- It also would require us to use only hapi.js-specific modules that would not be compatible with other frameworks, which could cause issues down the line.
- The main drawback to Hapi.js is that refactoring is all done manually, and endpoints are both created and tested manually [32].

4.3.4 Web Hosting

Web hosting is a server that will allow us to store our files on a remote server. This server will allow internet access to our files. This is needed, simply because we must store our files somewhere. With a web host, we can ensure our website is available to the public. Without it, our website would only be accessible on a local machine/network.

4.3.4.1 Amazon Web Services

Pros

- One of the main benefits of using AWS is the uptime. With the full power of Amazon, our application is almost guaranteed to always be accessible.
- AWS will also cover all our needs within the free tier that they offer [33].
- If/when our application needs to be scaled up, AWS offers scaled pricing based on how much data is used [33].

Cons

- AWS is said to have a difficult learning curve compared to other similar web hosts [7,8]. Furthermore, it is reported as user unfriendly.
- Another downside to AWS is the sheer amount of services it offers. This may sound like a pro at first, but most of these services have their own pricing and finding exactly what you need is not easy [34].
- Finally, the pricing of AWS is just confusing in general. With how many features it offers and a scaling cost, a consultant might be required to simply figure out how much you will need to pay [34].

4.3.4.2 Drupal

Pros

- Drupal would be a good choice because of how extensible it is.
- It's a blank slate that can defined exactly to user preference [35].
- Drupal is also excellent for building rapid prototypes [35]. This would allow us to attempt to implement multiple front-end designs/mockups with little time in between.

Cons

- However, to gain access to the "High flexibility" tier, there would be a large fee attached to it [36].
- In general, Drupal can have a large learning curve [35].
- It also requires a lot of effort to push any updates to the site [35]. We believe this would be problematic while we are designing and pushing major updates and features to our application.
- Drupal is also very resource intensive, and can drastically slow down the end user's speed down [35]. This is also something we can't afford to have since one of our main priorities is a highly reactive website.

4.3.4.3 Heroku

Pros

- Heroku is an extremely easy service to get started. An app can be deployed there in only a minute or two, by doing nothing more than a git remote and pushing to it [37].
- Heroku is also extremely extensible.
- The addons Heroku provide cover almost every need an application might need [37].
- Heroku is very flexible. Since it's UNIX based, you can essentially run anything you want/need to in their manage environment, then scale it appropriately [37].

Cons

- While Heroku offers a free tier, it is very limited. It can become extremely expensive as the application scales up to higher processing needs [37].
- The deploy time is also slow compared to other options. While the first deployment from git is fast, anything after that will be extremely slow [37].
- Everything on the server needs to recompile and restart, which results in downtime for the website/application [37].

4.3.5 Conclusion

For our structural framework, we have decided to use AngularJS. This is a middle ground between lightweight/flexible and robust/rigid. It also seems to offer the most features while also working well with our Node structure. The REST functionality will also be extremely useful for responsiveness. For our Server-side Web Application Framework, we decided to use Express.js. This is mainly due to the lack of documentation/support for Koa.js and the incompatibilities with hapi.js. We may end up using hapi.js as well for additional functionality. For web hosting options, we believe that AWS will be the best option. Its free tier is perfect for our project size. It also offers the best scaling, should our project grow to need it.

5 Weekly Blog Posts (all team members' posts)

5.1 Daniel Blog Posts

5.1.1 Fall 2017

5.1.1.1 Week 1

Plans:

- Get a usable MEAN stack app working
- Contact and meet with potential client
- Submit project preferences

Progress:

- Met with Jack Woods, got him to email a request for me

Problems:

- MongoDB is a pain
- Need to learn more about package managers
- Need to set up db
- Node for routes? Angular has routes too.

Summary:
This week was an introductory period where we went over the course outline and details in lecture. I began to pursue a project that I really wanted which is the Scalable Web Application for Energy on Campus. I met with Jack Woods and learned more about the project and requested that he submit a request for me to be on the team. I submitted project preferences with the MEAN stack application as my first choice.

5.1.1.2 Week 2

Plans:

- Get mocha js and d3 js incorporated in app
- Create archive of helpful/related resources

Progress:

- Got mean stack working completely

Problems:

- Team needs to get up to speed on MEAN stack web development

Summary:
This week we found out our groups for the project. We met and created a slack group for communication. We are making pushes in the right direction and sharing resources to get everyone on the same page. I built a template mean stack application that is a simple to do list from a tutorial online, it works.

5.1.1.3 Week 3

Plans:

- Get mocha js and d3 js incorporated in app
- Create archive of helpful/related resources

Progress:

- Got mean stack working completely

Problems:

- Team needs to get up to speed on MEAN stack web development

Summary:
This week was a good push in the right direction. We met with our clients on Tuesday and went over the general idea for the web application. I also set up the git hub repo and got everyone invited. Made directories and README. I didn't have much time to work on the test application but have been really dying to build up a template angular app with some d3 incorporated.

5.1.1.4 Week 4

Plans:

- Get mocha js and d3 js incorporated in app
- Finish Final Draft of the problem statement
- Get all requirements from client
- LEARN SOME REACT.js
- Have Kevin look over requirements

Progress:

- Kevin accepted the requirements list from Jack
- Almost finished with Problem Statement

Problems:

- Client is seemingly slow at responding to requests and emails -¿ need to address him and send him requests sooner so he has more time to respond.

Summary:
This week we finished the problem statement and sent them to our client Jack. We also requested that jack send us a rough draft of the requirements and features his team is expecting from us by the end of the term. We received these requirements and I met with Kevin to look them over. He said they seemed very reasonable and that we should go forth with minimal consultation/negotiation. I also created a mock up for the requirements document in LaTex.

5.1.1.5 Week 5

Plans:

- Get mocha js and d3 js incorporated in app
- Finish Rough draft of requirements
- Possibly make plan of action

Progress:

- Meeting with client at 5:30 on Tuesday
- Rough draft done
- Did not work on template with mocha and d3

Problems:

- I couldn't seem to get the Gantt chart latex package to work on os-class so I just added an eps. I will try to figure out how to use the package.

Summary:
This week we finished the rough draft of the requirements document and began working on the final draft. I plan to mess around with some MEAN stack stuff next week and try to get a d3 example web app going.

5.1.1.6 Week 6

Plans:

- Get mocha js and d3 js incorporated in app
- Finish Final draft of requirements
- Possibly make plan of action
- Start tech review? At least split up components

Progress:
- Finished final draft of requirements document
- Made plan of action for work throughout the term
- Split up components for technology review

Problems:
- Finally found a latex package that worked to create the Gantt chart our requirements document.

Summary:
This week we finished the requirements document and got it approved by our client. We had a helpful discussion in our TA meeting about generating a plan of action for assignments to follow and how we want to set up our workflow. We like communicating over Slack and have been doing fine working remote. We also began writing our technology reviews.

5.1.1.7 Week 7

Plans:
- Get mocha js and d3 js incorporated in app
- Do the tech review
- Review web sockets
- Make an app with web sockets that JSONifies?
- Get access from client to see the meter data from the AcquiSuites

Progress:
- Completed websocket personal application
- Implemented D3 dynamic charts with websocket application

Problems:
- I still want to figure out how we are going to get access to the data from the Acquisuite servers, and if we will be able to have it push to our EC2 instance once it is set up.
- I want to get this project started, but I think it all begins with the back end and handling data, which is still black boxed to us right now.
- Our client Jack was nice enough to look into what AWS package will work best and handle setting up an EC2 instance with the Office of Sustainability.

Summary:
This week we had a very helpful client meeting about the data acquisition servers and setting up AWS. On a personal accomplishment, I got D3 working on my web socket application, now I just need to make the d3 bar chart I found update based on the web socket data. We also began the tech review to be finished by Tuesday.

5.1.1.8 Week 8

Plans:
- Get mocha js and d3 js incorporated in app
- Get websocket data into d3 bar graph
- Finish the tech review

Progress:
- Completed tech review rough draft
- Implemented D3 dynamic charts with websocket application

Problems:
- I was researching hashing algorithms and authentication measures, but talked to Kevin and my TA who said I should just outsource authentication to google.
- Next week I will change my tech review from hashing algorithms to means if authentications.

Summary:
This week I finished the tech review rough draft and started really researching nitty gritty details about the dev ops behind this project. I spoke to my boss from my last job and got 3 page of notes about dev ops and total web application architecture. (I will up load this notes next week). I did not have very much time to work on personal MEAN stack projects this week as I was very busy with other work.

5.1.1.9 Week 9

Plans:

- Get mocha js and d3 js incorporated in app
- Get websocket data into d3 bar graph
- Finish the tech review
- Change authentication section to “means of authentication” in tech review
- Finish tech review final draft (early?)

Progress:

- Completed tech review final draft
- Changed authentication section to “means of authentication” in tech review

Problems:

- I was researching hashing algorithms and authentication measures, but talked to Kevin and my TA who said I should just outsource authentication to google.
- Next week I will change my tech review from hashing algorithms to means if authentications.

Summary:
This week I finished the tech review final draft early (by 1:00 pm) and emailed to Kirsten. We are still waiting on Jack, our client, to set up EC2 with the IT guys in his office. I worked on a lot of work for other classes to prepare for the design document this upcoming week. My group mates have been nonchalant with the work load and the amount of work that is required for the rest of the term’s documents. I had no time to do the personal projects I wanted to do, but freed up time for next week to work on stuff.

5.1.1.10 Week 10

Plans:

- Get mocha js and d3 js incorporated in app
- Finish the Design Document
- Make a template for Google auth with passport.js

Progress:

- Completed the design document
- Made a sample application with Google oAuth2.0 with Passport.js

Problems:
Began to think about the full architecture of the application brought up some design constraints that will need to be addressed before implementation. A lot of components and individual model objects will need to share data in order to render the correct graph and content to the page which will require explicit data model designs that are consistent and robust. Creating Angular services to process the data from multiple data models and components will be essential, but our application needs to ensure that these services receive consistent data in the forms it is expecting. An example of this will be the timestamped data entries logged in each building will need to be parameterized the same way for every building and type checking will need to be done in the service.

Summary:
This week was a grind through the design document and the cataclysm of other projects and assignments due for other classes. I was able to get a very functional MEAN stack application running that includes Google oAuth 2.0 authentication with passport.js middleware and effectively stores user profiles into user objects and maintains user sessions. I was also able to dive into the realm od new Bootstrap 4 components and design an HTML template for our block components using bootstrap 4 “Cards” which are the new take on “well” elements. Cards provide a container with a header row and content row that can flex to the size of its elements. Along with the bootstrap grid layout system, the two mingle very well to create the basis of what we need for our dashboards. In all, creating a design template application really allowed me to grasp the different components of the entire application and develop design strategies to make these modular subsystems work together seamlessly.

5.1.2 Winter 2018

5.1.2.1 Week 1

Plans:

- Make plans for the term

Progress:

- None

Problems:

- None

Summary:
Introductory week. We got settled back into the groove of things and began planning our attack on the project.

5.1.2.2 Week 2

Plans:

- Set schedule for work
- Get repo on Ec2
- Continue to work on weekly blogs

Progress:

- Made substantial progress on the application. Most progress since break.
- I got authentication working and users stored in the DB based of Google tokens.
- I got the block "create" form to auto-populate and move building objects from the drop down to the list view of "Selected Building."
- Added a demo of create mongoose functions to add buildings to the database and the building schema in our models directory.
- I assigned issues on github and got them synced with our waffle.io agile workflow program.
- Began using a .env file for sensitive tokens and id's. Using the dotenv npm library to use them in the code source.

Problems:

- Would like more confidence from teammates with web development proficiency.

Summary:
This week was much better than week 1 in terms of progress. We met with our client and set up a good plan for moving forward including using waffle.io for issue tracking and swim lanes. I pushed some good functionality that demonstrates how to go from HTML to controller to service to database and populate the view with the necessary data. Hoping this push will be used as a platform for moving forward. I got Google authentication to work and had our client create a google account for the Office of Sustainability to get a GOOGLE API token to use. Also got secrets and ID's protected in a .env file, good practice. Moving forward I plan to do the same thing I did with create block to create dashboard and create story to lay the foundation for the main components of our application.

5.1.2.3 Week 3

Plans:

- Set schedule for work
- Get repo on Ec2
- D3
- Dashboard functionality
- Story functionality
- Acquisuite data (websocket server)
- EC2 establishment.

Progress:

- Imported building images.
- Made mongoose functions for buildings.
- Finally figured out the populate mongoose method which gathers documents based on ID in a relational database.
- Was able to completely populate block components with array attribute fields and accomplished a nested ng-repeat angular data bind to display all buildings within a block within a users block array.

Problems:

- Making MongoDB function as a relation database was more difficult than expected.
- Finally figured it out with hours of reading documentation.

Summary:
I made great progress on a personal note to our application. Finally figured out the "populate" mongoose.js method which gathers subdocuments based on ID. I got building objects to populate in 'blocks' finally. It required the mongoose.js function "populate" to get the document when stored by id. Functionality includes a nested ng-repeat directive in blocks.html and some database queries in the route handler.

5.1.2.4 Week 4

Plans:

- Set schedule for work
- Get repo on Ec2
- D3
- Dashboard functionality
- Story functionality
- Acquisuite data (websocket server)
- EC2 establishment.

Progress:

- Dashboard functionality is working
- Our repo is on EC2 and we have a data server receiving POST requests from Acquisuites
- Got our client to set up a data server and input our EC2 IP address on a couple of meters around campus.
- We are currently receiving meter data every 15 minutes.

Problems:

- Waiting on teammate to create data model for data timestamp model.
- Client is seemingly slow at responding to requests and emails -¿ need to address him and send him requests sooner so he has more time to respond.

Summary:
I continued pushing functionality for the major components of the application. Blocks and dashboards are implemented. I'm getting a good grasp on mongoose.js and AngularJS.

5.1.2.5 Week 5

Plans:

- D3
- Dashboard functionality
- Story functionality

Progress:

- Kind of a slow implementation week.
- Began on poster and started assigning presentation tasks.
- Worked on dashboards and stories.
- Met with Kirsten.

Problems:

- None

Summary:
Made good progress towards required assignments this week (i.e. poster and presentation); full confidence we will finish by due dates. Would really like to just get D3 over with. Might assign a day next week to sit down and just DO IT. Almost finished with dashboards (still need a view/delete). Going to begin on stories next week.

5.1.2.6 Week 6

Plans:
- D3
- Story functionality
- Finish Poster
- Finish Presentation

Progress:
- Did the progress report and presentation.
- Assigned turn in to other partners as I have been handling a lot of the work lately.

Problems:
- Video concatenation took longer than expected. Progress report was not uploaded by midnight of due date.

Summary:
This week was one of the most frustrating weeks of my life. No more to report.

5.1.2.7 Week 7

Plans:
- D3
- Story functionality
- Edit/Update functionality for all forms

Progress:
- Minor tweaks to application.

Problems:
- Still waiting for data collection to be completed by partner.
- Tried to get edit/create dynamics to work for a long time this week but could not figure out UI router/resolves.

Summary:
This was a slow week, I began to set deadlines for partners and myself so that we begin pushing out minimum viable features. Need to figure out how to pre-populate web forms when a user wants to "edit" a particular component. I looked at AngularJS routing with resolves, but nothing seems to work.

5.1.2.8 Week 8

Plans:
- D3
- Story functionality
- Edit/Update functionality for all forms

Progress:
- All main components are implemented except for graphs.
- I took meters and building relationships into my own hands because they were not getting done.

Problems:
- Lack of contributions.

Summary:
This week was very, very productive! Completed a lot, made plans to complete more, trying to assign simple helpful tasks to partners to just get things done.

5.1.2.9 Week 9

Plans:

- D3 –¿ Chart.js?
- Finish data retrieval
- Begin Charting

Progress:

- Data retrieval from Acquisuites seems to be working.
- Parker got a chart template into the application using chart.js
- Authentication hiding objects on home screen working with ng-hide directive

Problems:

- Data retrieval has had a lot of hidden bugs and unknowns that has made data collection and aggregation very difficult.

Summary:
At least we made progress. A lot of Parker and I's work on charts has been hindered by the lack of data retrieval. Data retrieval seems to be in order and we have tons of data in the database, so I'm excited to keep building. The format and process of meter data being sent is largely black boxed to us and our clients, so we have been incrementally fixing problems that arise.

5.1.2.10 Week 10

Plans:

- Finish charting
- Set up email service
- Filter data in graphs
- Finish building pages
- Export to csv
- Finish other small requirements

Progress:

- I have finally found a solution for view/edit functionality using AngularJS ng-init directive
- This allows me to call scope functions before the form and inputs are rendered to DOM.
- I wrote my essay early so I plan to turn that In per request by Kirsten

Problems:

- Data retrieval has had a lot of hidden bugs and unknowns that has made data collection and aggregation very difficult.

Summary:
We are very close to being feature complete. I'm almost finished on my paper and need to finish features and make demo video. I really hope that Ben (TA) was truthful when he said winter reports will be evaluated and contributions will be taken seriously. This has been a very difficult term.

5.1.3 Spring 2018

5.1.3.1 Week 1

Plans:

- New Data Model for Data Entries
- Reformat Design Doc
- Buffer date values with 0's
- Fix Date Filters for csv
- Complete fixes for date filters

Progress:

- Set TA schedule time
- Filtered chart data to get consumption PER day
- Filtered data to get 23:45 - 00:00 every day for daily consumption.
- Figured out that charts were being sorted when getting added to the chart JS object, fixed.
- Began storing Acquisuite meters with appended ”address” serial numbers.
- Created dynamic getter function for google map polygon coordinates

Problems:

- Did not understand why there were so many meters per Acquisuite server
- Date filter system is buggy. Need to fix when a week rolls into another week.

Summary:
Getting back into the swing of things after spring break. We have new information about the AcquiSuites and how building meters are set up. We will have to begin summing and aggregating data from multiple meters to get total consumption for certain buildings. Not the best news this late in the year.

5.1.3.2 Week 2

Plans:

- New Data Model for Data Entries
- Reformat Design Doc
- Continue to debug AcquiSuite data and large spikes that are occurring.

Progress:

- Transfer data aggregation functionality into mongoDB query.
- Major debugging of Acquisuite XML data and data in Database.
- Expo registration complete, release docs filled out and turned in.

Problems:

- Did not understand why there were so many meters per Acquisuite server
- There are weird values coming in ever so often from AcquiSuite servers.
- AcquiSuite’s are pushing data entries with timestamps at XX:XX:01, we have not seen this before.

Summary:
This week we had a good push in the right direction towards better understanding the data being received by the AcquiSuite’s and how we should collect and store them. Our client gave us new information about meter addressing and how the data acquisition servers are set up that is requiring us totally re-write our backend and data retrieval implementations. Additionally, we will need to find ways to sum and subtract a number of meters together to get the correct values for specific buildings.

5.1.3.3 Week 3

Plans:

- New Data Model for Data Entries
- Reformat Design Doc
- Begin WIRED Article

Progress:

- Began working on the WIRED article for class.
- Interviewed friend from Ninkasi Brewery System group
- Partitioned McNary data into McNary Complex and McNary Dining through a complex data aggregation process.

Problems:

- Still trying to wrap our heads around AcquiSuite format

Summary:
This week I spent a good amount of time trying to update our data retrieval process to be more reliable and more efficient. I want to perform summations and date filtering from the MongoDB query itself with Mongo’s native aggregation functionality, but it has turned out to be a difficult process.

5.1.3.4 Week 4

Plans:

- New Data Model for Data Entries
- Reformat Design Doc
- Finish WIRED article
- Finish Poster
- Begin Progress Report
- Begin documenting all functions and files

Progress:

- Finished WIRED
- Worked a lot on the new charting algorithms
- Summation of multiple meters and filtering by date
- Got the poster done, I like how its evolved.
  We added screenshots, captions, group photo, and reduced some text into bullet points.
- Now performing "daily" calculations in the backend to reduce computation after request response.
- Took group photo for poster Continued working on back-end data aggregation and charting

Problems:

- Still trying to wrap our heads around AcquiSuite format

Summary:
I finished my WIRED article and submitted the Poster for final review. We took a nice picture in Kelley engineering, updated some graphics, and edited some of the text. I also updated the building data retrieval service to perform a lot of logic in the query prior to doing manual aggregation after receiving all the data entries. I think this will increase performance.

5.1.3.5 Week 5

Plans:

- New Data Model for Data Entries
- Reformat Design Doc
- Finish Progress Report
- Complete meter subtraction for monitoring McNary residence hall

Progress:

- Marked where all requirements are met in our code
- Continued working on back-end data aggregation and charting
- Submitted code snippets with Carbon to create nice screenshots in presentation Made progress report and presentation

Problems:

- Having difficulties finding an efficient way to perform complex data aggregation

Summary:
Spent a large portion of this week on the spring midterm and progress report. I continued developing data aggregation queries to match the new format we are receiving and storing data. Instead of performing a lot of for each function over and over to perform subtractions and different filtering, I'm trying to do as much as possible in the Mongo query so that we can reduce runtime. McNary hall is the difficult one to accomplish. One building needs data subtracted from two meters (i.e. it does not have its own energy meter, but rather is the difference between McNary complex consumption and McNary dining consumption). Not exactly the most intuitive calculation to perform in a Mongo query.

5.1.3.6 Week 6

Plans:

- New Data Model for Data Entries

- Reformat Design Doc
- Finish for code freeze

Progress:
- Updated UML for design document
- Got entire code source finished for code freeze
- Finally finished the McNary hall subtraction issues

Problems:
- None. Good Week.

Summary:
This week we finished implementing and testing our code before the code freeze on Friday. We had a couple small bugs and small implementations to finish, including McNary data aggregation, permissions based access, and UI bug fixes. All in all, a very productive week.

5.1.3.7 Week 7

Plans:
- Prepare for Expo
- Get demo video prepared for Expo demonstration

Progress:
- Created Expo demo video

Problems:
- None.

Summary:
This was Expo week, no more need be said.

5.1.3.8 Week 8

Plans:
- Begin working on final assignment.

Progress:
- Nothing.

Problems:
- None.

Summary:
Did not work on very much this week. Plan on getting a big push on the final paper and presentation next week.

5.1.3.9 Week 9

Plans:
- Begin working on final assignment.

Progress:
- Compiled a majority of the final tex assignment
- Defined new command for subsubsubsections as we will need them for report
- Using “input” command and temporary tex files to keep the report modular and cleaner

Problems:
- Using the bibliography with the temporary files was a struggles. Needed to clear aux files and recompile.

Summary:
Made a great push towards finishing the final report. Plan on being done by next week and have finals week totally clear.

5.2 Parker Blog Posts

5.2.1 Fall 2017

5.2.1.1 Week 1

Plans:

- Decide project, set preferences
- Keep updated with announcements
- Update resume as needed

Progress:

- Project prefs set and awaiting official project assignment

Problems:

- None so far

Summary:
Read through all project descriptions and chose my preferences.

5.2.1.2 Week 2

Plans:

- Get project assignment
- Meet client
- Meet group members
- Hash out objectives and details of project
- Set up communication with team and client
- Figure out a work plan as well as work environments

Progress:

- Completed all propsed goals for the week

Problems:

- A big issue that occurred is our meeting time with our client is after our assigned problem statement essay. This will make it difficult but not impossible to come up with an accurate problem statement. Although a brief rundown was given by the client, specific details have not yet been discussed.

Summary:
Communication between clients and group members has been established via Slack and other applications. Group has become more organized and meetings are scheduled for client. TA meeting times recommendations have been scheduled, no official meeting time has been decided. We still need to figure out our work plan and what programs we will be using to build our project. We also still need to meet with client, which is scheduled for Tuesday 5:30-6:30, and hash out specific requirements of the project.

5.2.1.3 Week 3

Plans:

- Finally meet with client
- Turn in problem statement
- Establish git hub
- Meet with TA

Progress:

- Met with client and associates. Discussed plans for meetings, as well as project requirements. Should be receiving a detailed list of website feature priorities within the week.

Problems:

- Illness

Summary:
Our meeting with our client was very useful. We have established that our client will provide us with a list of features that they want on the website in order of priority. It will contain ”must have’s”, ”would be nice to have’s”, and ”stretch goals”. It also outlined further what the website structure should be.

5.2.1.4 Week 4

Plans:

- Clarify goals of project from client
- Deliver final draft of problem statement for class

Progress:

- Met with TA to go over general project requirements and work processes. Our TA believes that this project is feasible in the amount of time given to produce it and is modular enough to keep us busy for the duration of the project.
- We have requested and received a detailed list of requirements by the client so we may have a more clear project statement, project statement final draft has been completed and has been sent to the client for approval by Friday!

Problems:

- Client is seemingly slow at responding to requests and emails.

Summary:
We have clarified the process and requirements of the class once more with the TA, we have also constructed our final draft of our problem statement so that everyone has a clear understanding of what is needed for this project to be considered complete.

5.2.1.5 Week 5

Plans:

- Complete requirements Document rough draft
- Meet with client to further solidify end project requirements
- Learn about MEAN stack development

Progress:

- draft completed
- Met with TA and client

Problems:

- None

Summary:
We have completed our requirements document rough draft and solidified project requirements and stretch goal

5.2.1.6 Week 6

Plans:

- revise rough draft and Get started on final draft of requirements document
- learn more about MEAN stack development and agile workflow

Progress:

- Section 2 of requirements document has been revised by instructor and edits have been made. The section has been made more general with information and less specific as requested by the instructor
- MEAN stack development tutorials have been started

Problems:

- None

Summary:
Continued to learn mean stack development and create mental plans for application. Requirements doc completed and turned in.

5.2.1.7 Week 7

Plans:

- Discuss technical document requirements and assign technical analysis to each team membe
- Meet with client to discuss database information for our application

Progress:

- Info about technical document discussed, specific technicalities not defined yet
- MEAN stack research continued

Problems:

- None

Summary:
We have met with our TA to get a better understanding of the requirements of the technical document rough draft. We have also met with our client to get more info on application database requirements, specifically which web hosting service we will use and how much data we need to store. Mean stack development has also continued to be researched.

5.2.1.8 Week 8

Plans:

- Read about node.js
- Read about comparisons between node.js, php, and others. Mongodb, mysql, and kassandra. And aws ec2, google ce, and azure cosmo db.
- Turn in tech review rough draft

Progress:

- Comparisons researched and technologies decided.
- Final draft tech review began and will be completed come next tuesday

Problems:

- Tech review with peer review was lost somewhere when being returned on thursday, but has been accounted for by the professor.

Summary:
This week I finished the tech review rough draft and started really researching nitty gritty details about the dev ops behind this project. I spoke to my boss from my last job and got 3 page of notes about dev ops and total web application architecture. (I will up load this notes next week). I did not have very much time to work on personal MEAN stack projects this week as I was very busy with other work.

5.2.1.9 Week 9

Plans:

- Eat a lot of turkey
- Create a schedule to complete upcoming assignments
- Finish the tech review

Progress:

- Schedule for upcoming assignments created

Problems:

- Ate too much turkey

Summary:
I ate a lot of turkey and got started thinking about what I will need to do to complete the upcoming design document as well as the progress report. A schedule has been made that includes these assignments as well as assignments for other classes.

5.2.1.10 Week 10

Plans:

- Finish design document by Wednesday
- Get started on progress report on Friday or this Saturday

Progress:

- Design document core intro section completed, components assigned
- Design doc completed, proof read, and submitted

Problems:
none **Summary:**
We worked hard to get our full design of our application figured out and put into an organized design document. We completed and proof read our document to catch as many grammatical errors or errors is definitions as we could and submitted the document for grading. Progress report work will be postponed to Sunday or monday

5.2.2 Winter 2018

5.2.2.1 Week 1

Plans:

- Go to class and to establish foundation of future term

Progress:

- Class will be only at announced dates for specific info
- Mean stack, specifically angular studied

Problems:

- EC2 instance not up and running yet, development will be local for now

Summary:
Our while we can work on our application locally, our EC2 instance is still not up and will need to be handled by our client. Once that is done, we will have a working public access link to build our website on

5.2.2.2 Week 2

Plans:

- Format the ui of the contacts and about page of the top nav section.

Progress:

- Contacts and About page was formatted and social media for the Sustainability office have been added

Problems:

- Our EC2 instance is still not up but should be up this week or next.
- Google API not functioning and cannot figure out why

Summary:
Our sites production is coming along nicely. While we still do not have a backend, we have a template for most parts that will utilize the backend

5.2.2.3 Week 3

Plans:

- Fix google maps Api
- Update hard coded CSS to our CSS file
- Set up EC2 instance and credentials

Progress:

- Continued work on Google maps API solution
- Currently searching for mean stack single application references to use

Problems:

- Google Maps API is still broken

Summary:
CSS has been transferred from hard coded elements to the CSS file and our EC2 instance has been set up by our client so our application can be used publicly. I am struggling with getting the google maps API working but I have not exhausted all options yet

5.2.2.4 Week 4

Plans:

- Fix Google Maps API
- Get Fonts from OSU marketing
- Fix contacts and about page UI to look nicer

Progress:

- I have found a resource to fix the google maps API
- Google Maps API implemented and working

Problems:

- Backend framework seems to be slightly delayed with updates.

Summary:
currently the google maps API is now functional but still needs the KML layer to allow for users to select buildings. Progress is being made on our backend framework and likely we will be getting live updates of data within a couple weeks. The contacts and about page has been reformatted to look cleaner

5.2.2.5 Week 5

Plans:

- Adjust top nav section
- Add side nav animations

Progress:

- Top nav has been reformatted
- Side nav now has animations when elements are clicked

Problems:

- None

Summary:
The top navigation bar of the application now has a main title in the center as opposed to the left with an OSU approved font. It redirects to homepage when selected. The logout button has been reformatted. The CSS and styling has been reformatted to be a fixed size. The official OSU logo has been added

5.2.2.6 Week 6

Plans:

- Get OSU approved Fonts
- Get OSU approved logo customized for our site
- Implement KML layer to google maps
- Finish Poster
- Finish Presentation

Progress:

- OSU fonts obtained as files and and implemented
- Logo requested

Problems:

- Client response times slow, may need to message them again

Summary:
We have a logo that does not necessary fit with the style of our website. We have requested to our client that he contact OSU marketing to get a customized logo for our application but I have not heard back yet. We have obtained the fonts for OSU approved fonts and have implemented them into the headers of the site. We also need to implement the KML layer of the google maps soon

5.2.2.7 Week 7

Plans:

- Complete google maps API
- Adjust dashboard CSS

Progress:

- Google maps API 90 percent complete
- Dashboard CSS complete

Problems:

- Google Maps API cannot update pages via URL, need to create new controller to get building objects
- Backend still in a relatively young state

Summary:
Our backend is taking shape but is perhaps behind schedule, we are not retrieving any data to be put into graphs yet but seem to be getting close. Our google maps api for our homepage is almost complete but I need to figure out how to pass specific building objects to get our building pages, a simple URL will not suffice, the data must be retrieved. The dashboards CSS has been adjusted to include a description

5.2.2.8 Week 8

Plans:

- Finish Google Maps API
- Help create controllers for data retrieval for dashboards

Progress:

- Data retrieval has been handled by aubrey and Jack.
- Finished styling on all current pages and form pages.

Problems:

- google maps API will be postponed as a stretch goal.

Summary:
Pretty much the whole week was dedicated to styling every page to make sure it has proper font, scheme, and element positioning.

5.2.2.9 Week 9

Plans:

- Add authentication and charts to site
- Assign these roles accordingly and make set dates for these to be completed

Progress:

- Added template chart that is functional but needs to populate on more elements rather than just 1 and use data retrieved from our DB
- Assigned me to charting tasks and daniel to authentication tasks

Problems:

- Backend took a while to get complete, so progress on creating charts has been limited until now

Summary:
Authentication has been completed by Daniel and I have completed the implementation of charts using Chart.js. These charts still need to have the ability to filter by date and filter using different data types but overall the API has been implemented.

5.2.2.10 Week 10

Plans:

- Implement Chart data filters
- Implement email authentication
- Implement data export to cvs files
- Complete progress report and demo video by finals due date

Progress:

- Charts have data filters now
- Email API has been implemented, no authentication ability yet.

Problems:

- Finals week makes for difficulties to find time to complete all tasks.
- Charting API has some bugs when hovering over data points

Summary:
Our project is essentially coming to a close. We are almost entirely done with the application with a few small features missing. These features should not take very long to implement. Our charts now have more filters and displays on the buildings page correctly. Email API has been implemented but does not yet have the ability to set administrative privileges. My progress report is almost complete and the demo video should be complete by the due date.

5.2.3 Spring 2018

5.2.3.1 Week 1

Plans:

- Get future term plans ironed out
- Figure out TA meeting times

Progress:

- Generalized plan has been made for the future terms
- TA meeting next week.

Problems:

- None

Summary:
Introductory week as given a good foundation for future plans, not much has gone on with regards to assignments or project.

5.2.3.2 Week 2

Plans:
- Fix any UI structure inconsistencies for application

Progress:
- Adjusted some small navigation tool issues.

Problems:
- Not much to fix with regard to UI.

Summary:
Aquisuite behavior was strange but Daniel and Aubrey seem to be making progress with regards to that. UI tweaks have been made to the site.

5.2.3.3 Week 3

Plans:
- Get signatures for expo
- Figure out why some entries in application are being strange

Progress:
- All documents for expo signed by group members to allow us to attend

Problems:
- Complex and unforeseen issues with how aquisuite data is formatted. Will need to reformat and rework our data organization.

Summary:
More minor UI tweaks and preperation for expo was basically the summary of what his weeks content was.

5.2.3.4 Week 4

Plans:
- Finish getting poster fully complete
- Get office of sustainability head shots as well as team headshots for the contacts page.

Progress:
- Most headshots have been acquired. Missing some still.

Problems:
- Communication issues preventing contact page from being complete

Summary:
contacts page has been formatted to the specifications of Brandon from the Office of Sustainability.

5.2.3.5 Week 5

Plans:
- Get our site tested by our client or the office of sustainability.

Progress:
- Communication and goals to get our app tested has been established
- Client agreed to sit down with members of the office of sustainability for testing.

Problems:
- Delays in scheduled testing time
- Still no application walkthrough has been made by client.

Summary:
Prepping for expo is made slightly more challenging as our client has not had the chance to look over our project for testing and confirmation that it meets expectations.

5.2.3.6 Week 6

Plans:

- Fix google maps implementation so buildings will either redirect to default building page or to the associated buildings page.

Progress:

- Correct buildings are now defaulting to the main buildings page.

Problems:

- Communication issues caused difficulties clarifying the goals of the google maps UI. Still deciding on what to do,

Summary:
Finished up final changes for code freeze before expo. Nothing much else has occured.

5.2.3.7 Week 7

Plans:

- Prep for expo
- Iron out any last minute changes and pull requests.

Progress:

- Expo complete, went well.

Problems:

- Minor PR changes did not get made prior to expo
- Slight connection and version issues during expo but overall did not reduce presentation quality.

Summary:
Expo week went well and many people were impressed by our project.

5.2.3.8 Week 8

Plans:

- Iron out any last minute application changes.
- Communicate with team on upcoming assignments.

Progress:

- Communication established. Application seems to be in final format.

Problems:

- Not much left to change with regard to application UI.

Summary:
Application finalized.

5.2.3.9 Week 9

Plans:

- Figure out all deadlines for any upcoming assignments
- Format weekly blogs
- Figure out a date and method for the application handoff

Progress:

- Deadlines understood. Plans made for completing upcoming assignments.

Problems:

- Senioritis

Summary:
Final report progress has been made

5.3 Aubrey Blog Posts

5.3.1 Fall 2017

5.3.1.1 Week 1

Plans:

- Submit project preferences

Progress:

- Decided on my top project choices
- Researched different projects to figure out what I wanted to do

This week was an introductory period where we went over the course outline and details in lecture. We were told to select a few of our top choices for projects.

Problems: No problems to report.

Summary:

5.3.1.2 Week 2

Plans:

- Get together with team and talk with client
- Go over project scope

Progress:

- Got everything initialized between team and client

Problems:

- Nobody seems to know how we are going to implement all of this

Summary:
This week our team set up our Slack channel for communication purposes. We also were able to get a hold of our client to schedule a meeting for the next week. Got access to current implementation of web interface we will be replacing.

5.3.1.3 Week 3

Plans:

- Work on problem statement assignment

Progress:

- It was good to finally meet our client and discuss our project in more detail.

Problems:

- After our client meeting, we had a general sense of what would be delivered as a final product. However, we still had no idea how or what we were going to do to implement it. One of the main problems that arose from the meeting was the discussion of hardware for our project. We had no idea where we were going to store our database and website. As a solution, Jack told us he would talk to his supervisors and IT group to figure it out.

Summary:
This week involved a general setup of resources for the rest of the term including our Github repository and our OneNote Notebooks. We also met with our client, Jack Woods, at his office in the Office of Sustainability on campus to discuss details about the project and their visions for the web application.

5.3.1.4 Week 4

Plans:

- Start on requirements doc

Progress:

- Finished the Problem Statement final draft and submitted on Github and OneNote.
- Emailed our client requesting a list of requirements he and his team want to see in the final application to begin working on the Requirements Document assignment.
- This was the first week we were required to meet with our TA, Ben, where we discussed the structure and organization of directories in our group repository and how to create a logical labeling system.
- Made a plan of action to begin researching the MEAN stack and its different frameworks.

Problems:

- Issues with LaTeX formatting and compiling, but was solved eventually.

Summary:
Finishing the Problem Statement doc helped our team develop a general sense of the project.

5.3.1.5 Week 5

Plans:

- Finish up Final Draft of Requirements doc
- Learn about more about MEAN dev

Progress:

- Got a rough draft of the Requirements Document completed and submitted on Github and OneNote.
- Met with our client to discuss further details about the components listed in his requirements list.
- Our client also said he mailed his IT group about potential databases and servers that could be used for our application, but they had not responded yet.

Problems:

- We had some severe issues with creating a Gantt chart into our document. The Gantt LaTeX package just would not work properly.
- We also had a few issues with Github merge conflicts.

Summary:
Requirements doc further solidified our understanding of the project.

5.3.1.6 Week 6

Plans:

- Create and mess with simple MEAN app
- Research Database and website hosting.

Progress:

- We got a Latex Gantt package to work in our Requirements Document and submitted final draft to Github and OneNote.
- Contacted our client to review and approve our Requirements Document and send a confirmation email to Kirsten Winters and Kevin McGrath.
- Our group assigned three components to each member to research for our individual Technology Review documents.
- TA meeting revolved around clearing up the final questions about the Requirements Document and discussing our current actions required model.

Problems:

- No problems to report

Summary:
We were very content with the communication and rate of response from our client; it simplified our workflow and made accomplishing our tasks very trivial. Finishing requirements doc helped us immensely in figuring out what needed to be done for rest of project.

5.3.1.7 Week 7

Plans:

- Meet with client to go over hosting services

Progress:

- Did heavy research into webhosting and database options for the project.
- This was all preliminary research done ahead of time to present to our client for a meeting next week. I specifically focused on finding information on Amazon AWS.

Problems:

- We have had trouble getting our client and his office to set up an AWS account to host our application.
- He ensured us that he would get it taken care of, but we are currently stuck with only local development strategies in the meantime.

Summary:
Doing research into AWS will help us quite bit, both for our meeting next week and tech doc.

5.3.1.8 Week 8

Plans:

- Begin design doc

Progress:

- This week our group finished the individual Technology Review rough drafts and began working on the final drafts for next week.
- Underwent extensive research on specific technologies that would be incorporated in our application and generated good documentation of examples and resources to help drive future development.

Problems:

- We had some issues trying to figure out how to handle security.
- After talking to Kevin and Ben, we decided to outsource it to a third party rather than making it ourselves.

Summary:
This week I finished the tech review rough draft and started really researching nitty gritty details about the dev ops behind this project. I spoke to my boss from my last job and got 3 page of notes about dev ops and total web application architecture. (I will up load this notes next week). I did not have very much time to work on personal MEAN stack projects this week as I was very busy with other work.

5.3.1.9 Week 9

Plans:

- Complete design doc

Progress:

- Submitted Final Draft of Tech Review
- Made template for Design Doc

Problems:

- The IEEE format for the design doc was hard to understand and we needed to talk to Kirsten about it.

Summary:
Very short week due to holiday.

5.3.1.10 Week 10

Plans:

- Design and Code project over break.

Progress:

- Wrote and submitted design doc

Problems:

- We had some troubles with explicit definitions of terms in our requirements from our client.
- We had multiple conversation and messages from our client where he used the terms “dashboard” and “page” interchangeably. After meeting with him, we cleared up the confusion and were able to establish a finalized dictionary of terms.
- Another issue we faced was the design of data models that presented themselves in the Design Document and determining how we were going to efficiently store and manage our different data and entities.

Summary:
Design doc finalized our understanding of project

5.3.2 Winter 2018

5.3.2.1 Week 1

Plans:

- Set up times for TA meeting
- Set up Client meeting

Progress:

- None

Problems:

- None

Summary:
Getting back on our feet after break.

5.3.2.2 Week 2

Plans:

- Make sure our EC2 instance works

Progress:

- Had first TA meeting of term
- Briefly went over break
- Had Client Meeting

Problems:

- None

Summary:
Setting plan of action.

5.3.2.3 Week 3

Plans:

- Wait for client to get sample data to begin backend

Progress:

- Found packages to use on backend

Problems:

- A lot of hardware problems with my computer set me behind schedule quite a bit.

Summary:
Continuing to build plan of action. Waiting patiently for client to get back to me so I can start on backend data collection.

5.3.2.4 Week 4

Plans:

- Work on backend parsing

Progress:

- Working on buildings.
- Got templates working fully

Problems:

- Very bad health issues left me incapacitated

Summary:
Finally could get some work done to server.

5.3.2.5 Week 5

Plans:

- Continue implementing backend of our application.
- Implement duplicate prevention.

Progress:

- Pushed out a bunch of updates to the backend.
- Our software can now read and add entries to database.

Problems:

- Had some conflicts with team dynamics, but we resolved those quickly.

Summary:
Huge server changes made.

5.3.2.6 Week 6

Plans:

- Fix duplicate prevention

Progress:

- Pushed out a bunch of updates to the backend.
- Finished progress report.

Problems:

- Having issues with my duplicate data check not comparing format correctly.

Summary:
Continued implementing backend of our application.

5.3.2.7 Week 7

Plans:

- Implement addMeters
- See if I can get addBuildings to work

Progress: This week I worked a lot on routing to make it easier to access schema data after POST data comes in.

- Implemented functional routes to add data to new schema
  - This involved changing the structure of the DB access order
  - Used to look for building-¿meter-¿entry
  - Now looks for meter-¿building-¿entry
- Changes made makes it easier to find necessary data without having to make another call later
- Changed HTML/controller files to match new schemas

Problems:

- None

Summary:
Decently busy week. Not much code went out, but it sure does a lot.

5.3.2.8 Week 8

Plans:

- Get LIVE data working on EC2 instance
- Finish implementing addMeter from XML POST route
- Change 'Edit Meter' controller to add data entries to new building when added.

Progress:

- Changed backend functionality in how it adds data entries
- Changed building html pages to reflect above changes
- Implemented controllers for edit buildings (work in progress)
- Fixed 'Add Building' to remove meters from old buildings when added to new.
- Moved XML POST route to dataServer.js
- Fixed duplicate prevention

Problems:

- The 'addMeter()' function I was implementing will require some tweaking of the meter schema.
- As the flow of the 'receiveXML()' works right now, it finds:
  - 'meter-¿building with that meter-¿pushes data entry to that building'
  - 'addMeter()' will have a building set to null, so referencing it will crash program. There's no way for us to be able tell which building a meter goes to (at least I think) from just the XML data
- My solution will be to add a data entry array to the meter schema. I would just add a check:
  - If the meter's building is null, then the meter data entry array will push the entry
- Then, once the meter gets added to a building (via edit meter or edit building), then all the contents of a meter's data entries would be pushed into that building and popped out of the meter's data entries (to prevent duplicate entries if it were to change buildings)

Summary:
A lot of work was completed this week and we now have AcquiSuites posting to our EC2 Instance.

5.3.2.9 Week 9

Plans:

- Plans for next week is to finish up add/delete different components. Each of those should take a while.

Progress:

- Met with client and debugged connection to EC2 instance and discussed current affairs
- Controllers for buildings/meters
- Completely finished data parsing and functionality of adding entries to entries DB
- Updated DBSample.js to push dummy data to local DB for graph testing by creating function in DBSample.js that uses RNG to generate timestamps and readings then adds them to DB. This allows Dan and Parker to work on graphs.
- Implemented functionality to remove meters from old buildings when it is added to new buildings. This is a very important fix because it prevents data from going to the wrong building!
- Finished 'addMeter()' in dataServer.js. This function adds a meter if it doesn't currently exist in the database. It uses data from XML to do so.
- Modularized XML function with async function calls to do various thing. Implemented this in case server gets hit with two POST requests at the same time. If the function blocks completely, it might reject data, which in turn could cause data loss.
- split XML function into addMeter() and addEntry()
- addEntry() is what actually pushes and updates
- Fixed error that occurred when buildings were null when trying to add entries
- Implemented functionality to let us keep track of data when it comes to our server before a building/meter is added to it.
  - Creates meter for data entry. Then it sets that meter as the DataEntries' Meter.
  - This function is called when a building is added when a meter.building is null
  - Pushes all the the matching data entries to the building array
- Implemented functionality to handle different types of XML requests. Essentially just makes sure it's the log file and appropriately formatted before trying to do anything with it.

Problems:

- I ran into a LOT of problems this week after we migrated our project to the EC2 instance.
- XML POST requests weren't always formatted the same.
- XML POST data was just completely different than the sample data I've had all term.
- Specifically there was multiple ¡record¿ entries, which I've only seen the XML contain one at a time This threw how addEntry() worked completely out the window.
- These two problems along with some minor things took about a good 10 hours to resolve.
- To resolve these issues, I added a check mentioned above. This was an easy one.
- The second one was not so great to fix. I ended up having to wrap everything in a promise, create a DataEntry array and add each ¡record¿ to that. Then it was as simple as iterating through the array and adding to the database. A big time factor was figuring out exactly why it was crashing in the first place. I was able to pull some logs from the server, which helped. It also didn't help that if I had one ¡record¿, it wouldn't be considered an array of ¡record¿.

Summary:
I made a ton of progress this week and look forward to wrapping it up.

5.3.2.10 Week 10

Plans:

- Finish progress report
- Fix any features that don't work 100
- Start debugging some of the more high priority bugs

Progress:

- Removed and fixed merge conflicts someone pushed that caused data retrieval to return some interesting results.

- Implemented a functions that resets the building variable of DataEntry to null when that building is deleted. This will allow past/stored data entries to push correctly to that building if buildings were added incorrectly.
- Related to above, implemented a function that set the building variable of DataEntry to the building id when being pushed from a meter that was storing data entries. Previously, only new data entries whose meters were already added to a building would be set as such.
- Fixed functionality so when meters were deleted, they would remove themselves from a Buildings array of Meters.
- Implemented and fixed edit/delete functionality of Block, Building, Meter, and Dashboard components.
- Worked on application optimization in some of the functions (they were causing application to be quite slow).
- Implemented main feature that allows users to enter an email address and receive a link from our application that would allow them register.
- Furthermore, implemented the front side of this as well. Also implemented it to only show when user is not Logged in.
- Found temporary fix for our 'null' building bug that keeps popping up. We simply don't load that entry.
- Updated all backend dataServer log statements to provide much more useful.
- Fixed async functions that were causing data to populate strangely.
- Fixed issue that was causing app to crash when any of the components were interacted with but only had one or fewer buildings.
- Implemented main feature that prevents malicious code to be injected into our database.
- Formatted all of our code docs so they had consistent styling.
- Implemented main feature for the upcoming Kilowatt Crackdown event.

Problems:

- Null building bug seems to reoccur randomly and the cause of it isn't apparent.
- Visual bug when using date selector for Kilowatt Crackdown
- Registration only works with AWS verified accounts until our AWS is moved out of sandbox mode.
- A lot of front end crashes caused by code that wasn't tested with pre-existed components.
- Data in backend was messed up for a while when client used the MongoDB Compass app to add entries instead of our application. This prevented a lot of the necessary data and functions to fire and make things work correctly.

Summary:
Very busy two weeks after an extremely busy week 9. I got a lot of the main features in our requirements doc complete.

5.3.3 Spring 2018

5.3.3.1 Week 1

Plans:

- Figure out why McNary is acting so strangely.

Progress:

- Set TA schedule time
- Over the break, it seemed our server went haywire. We received even more extremely weird entries.
- I spent most of the week debugging the backend of the server to figure out the issue. This really just involved a lot of trial and error, with plenty of conditionals and log statements.

Problems:

- Lots of problems with only some of the buildings reporting weird issues. This is much worse than if ALL the buildings were reporting errors.

Summary:
Good progress for first week of the term. Some buildings are acting weird.

5.3.3.2 Week 2

Plans:

- Fix the email alert functions.
- Continue to debug AcquiSuite data and large spikes that are occurring.

Progress:

- During this week, I noticed that some of the entries were showing negative values. After talking with our client, he determined that this is a hardware issue caused by some of the wires crossing. To fix this, I simply added a statement that would take the absolute value of each entry.
- Added a function that would retroactively change all the negative values to positive. There were a few more 'one time use' functions added.
- Also resolved an issue with an async function causing EC2 to crash.

Problems:

- Negative values showing where positive values expected.

Summary:
Bug fixing most of the week.

5.3.3.3 Week 3

Plans:

- Keep adding to email functions as well as start on user account stuff.
- Finish WIRED Article

Progress:

- Began working on the WIRED article for class.
- Interviewed friend from The Board Room group
- This week involved moving some functions around to make more sense. Specifically, I moved our email alert functions to our data retrieval server, as this is where the bulk of the interaction would take place.
- Added more robust functionality to all of the email functions.
- Added functionality so all of these email alerts would occur automatically. This was implemented by checking the timestamp then checking if alert parameters were true. This prevents an email going out every time a data entry comes in.

Problems:

- Troubleshooting AcquiSuite still

Summary:
A lot of work put into email functionality.

5.3.3.4 Week 4

Plans:

- Debug above functionality and ensure it works before midterm presentation.
- Update schema and login functionality to accept new users permissions.

Progress:

- Finished WIRED
- This week I mainly focused on starting the user access levels. Specifically, this was for the email registration page that allowed admins to add new users with various permissions.
- Took group photo for poster

Problems:

- Troubleshooting AcquiSuite still

Summary:
User permission levels implemented.

5.3.3.5 Week 5

Plans:

- Debug above functionality and ensure it works before expo.

Progress:

- This week I mainly focused on updating the schema and functionality for login. Specifically this was for accounts invited through the email invitation link.
- After a user was invited, nothing really happened besides them getting an email. After this update, users will be added to the database.
- Added minor fix to form input blacklist. There was also a lot of work done for midterm presentation.

Problems:

- None

Summary:
Good functionality updates and worked on progress report.

5.3.3.6 Week 6

Plans:

- Debug login functionality further.
- Prepare for expo.
- Prepare for code freeze.

Progress:

- This week I mainly focused on figuring out syntax for JSDocs. This is going to be our main code documentation, so I really wanted to focus on learning the ins and outs of it.
- I also added JSDocs for basic typedefs.
- This was for both regular JS objects and custom created objects we use in our codebase.

Problems:

- JSDocs seems a little finicky at first, but after some tweaking, I can see it being a powerful documentation asset.

Summary:
Documentation for final started.

5.3.3.7 Week 7

Plans:

- Continue on documentation.

Progress:

- This week I mainly focused on cleaning up our site so it would look better for expo.
- The most important update I implemented was sitewide security. I found this to be a huge priority because we would be demoing our project on our live website. If someone were to wander to that site on their own device, they could wreak havoc on our databases.

Problems:

- None.

Summary:
Expo was great!

5.3.3.8 Week 8

Plans:

- Keep working on documentation

Progress:

- This week I mainly focused on working on documentation. I will have a large amount of code to push in next couple of weeks.

Problems:

- None.

Summary:
Documenting everything.

5.3.3.9 Week 9

Plans:

- Finish presentation.

Progress:

- This week I mainly focused on getting my blog posts up to speed with progress. I also started on some of the final presentation and report stuff.

Problems:

- None

Summary:
Documenting everything.

5.3.3.10 Week 10

Plans:

- Gather things together and finish documentation.

Progress:

- This week I mainly focused on more code documentation. I also wanted to start getting all of my documents together for the final report.

Problems:

- None

Summary:
Finishing everything up.

MONITORING ENERGY USAGE ON CAMPUS

WHY IS MONITORING ENERGY IMPORTANT?

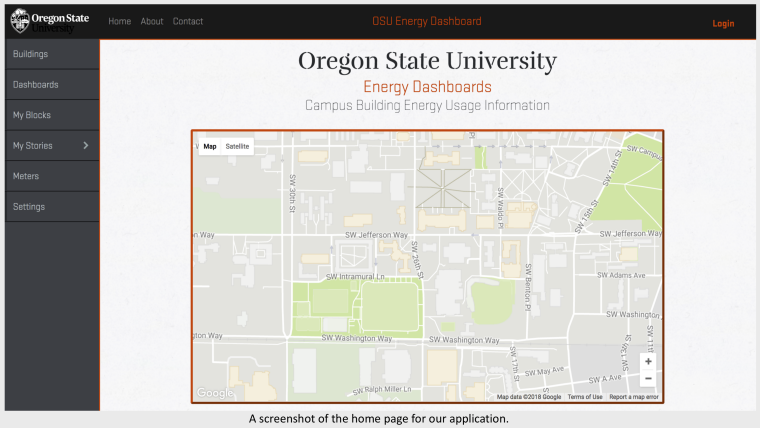
Oregon State University strives to be an energy conscious and energy efficient campus. In order to achieve this, The Office of Sustainability at Oregon State University continues to implement modern tools and initiatives to monitor energy usage data and use that data to make informed decisions on future infrastructure projects.

The OSU Office of Sustainability has installed energy meters in campus buildings to monitor and record energy data so that faculty and students can analyze the data and view trends about the consumption on campus. The goal of this application is to allow students and faculty at Oregon State University to visualize consumption patterns on campus and put quantifiable data to the abstraction of "carbon reduction." The Office of Sustainability has the overall intent to drive campus-wide energy use down in an effort to reach OSU's carbon neutrality goal of 2025.

HOW TO VISUALIZE ENERGY DATA?

In order to solve this problem, we are implementing our own application to monitor energy data from around campus. The AcquiSuite data acquisition servers send XML data to our application via HTTP POST requests. Our application parses and stores the data into a database.

Our application allows users to access the meter data and create dynamic charts and graphs on administrative-style dashboards. Users have the ability to generate and group multiple 'dashboards' into intuitive collections. This allows users to quickly traverse different data sets and generate reports quickly and efficiently.

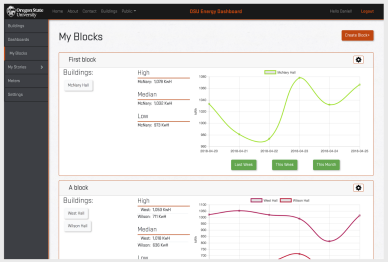


A screenshot of the home page for our application.

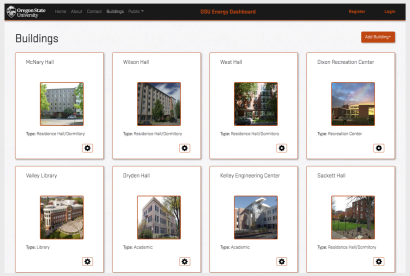
WHAT DID WE BUILD?

Our project was to create an administrative dashboard where users can generate energy reports for different buildings around Oregon State's Campus. Our application aims to visualize energy data in a meaningful way to display generic trends or comparative results for buildings and their energy consumption data.

- M.E.A.N. Stack
- R.E.S.T.ful
- Chart.js



A screenshot of user "blocks" and energy data.



A screenshot of the buildings page.

WHAT IS AN ENERGY MONITORING DASHBOARD?

An Energy Monitoring Dashboard is an application that generates and visualizes energy data. It will take raw data from different monitors and meters and use that data to create user friendly graphs and charts. Each meter can be categorized and displayed along other similar meters.

The purpose of an Energy Monitoring Dashboard is to provide the user with an intuitive way to monitor their energy usage. Instead of looking through spreadsheets of raw data, they can simply look at a graph to determine total/averages of energy usage.

Pictured: (left to right) Parker Bruni, Daniel Schroeder, Aubrey Thenell

MEET THE TEAM:

- **Daniel Schroeder**
  Applied Computer Science
  schrodan@oregonstate.edu
- **Aubrey Thenell**
  Applied Computer Science
  thenellb@oregonstate.edu
- **Parker Bruni**
  Applied Computer Science
  brunip@oregonstate.edu
- **Jack Woods**
  Client
  Office of Sustainability
  Utility Data Analyst Student Technician
  jack.woods@oregonstate.edu

Oregon State University

7 PROJECT DOCUMENTATION

7.1 How does one install your software, if any?

- In a shell, clone the repoistory via 'git clone https://github.com/DSchroederOSU/SeniorCapstone.git'

7.2 How does one run it?

1) Open two shell/consoles.
2) In both shells, Switch into the 'Source' directory
3) Install package dependencies via 'npm install'
4) Add your environmental variables to './config.js'
5) In your first shell, Start the data server via 'nodemon ./dataServer.js'
6) In your second shell, Start the public server via 'nodemon ./server.js'
7) In a web browser, navigate to 'localhost:3000' to access the application interface

7.3 Are there any special hardware, OS, or runtime requirements to run your software?

- If running in a local environment, MongoDB will have to be running on your machine

7.4 Code documentation

7.4.1 TypeDefs

Global

Type Definitions

`Array`

Source:	Documents/JSdocs/types.js, line 16
See:	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

`Block`

An object which holds customized combinations of Building as set by the User. Can be set to display on the public Dashboards or not.

Type:

- Object

Properties:

Name	Type	Description
`name`	String	The name of the Block as set by the User when created.
`created_by`	ObjectId	The ObjectId of the User responsible for creating the Block.
`building`	Array.<ObjectId>	The array of ObjectId references to Building that the Block contains.
`chart`	String	The type of chart used by this Block.
`is_public`	Boolean	The flag to determine if the Block should appear in the public Dashboard interface.
`variable`	String	The string to determine unit being measured (e.g. Killowatts/Hr).

Source:	Documents/JSdocs/types.js, line 66

`Boolean`

Source:	Documents/JSdocs/types.js, line 6
See:	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean

`Building`

An object which aggregates DataEntry as gathered by Meter

Type:

- Object

Properties:

Name	Type	Description
name	String	The name of the Building as set by the User when created.
building_type	String	The type of Building as set by the User when created.
meters	Array.<ObjectId>	The array of ObjectId references to Meter that are connected to the Building.
data_entries	Array.<ObjectId>	The array of ObjectId references to DataEntry that are connected to the Building.

Source: Documents/JSdocs/types.js, line 77

Chart

An object that contains the properties of Chart.js

Type:

- Object

Source: Documents/JSdocs/types.js, line 42
See: https://www.chartjs.org/docs/latest/

Dashboard

An object which contains Blocks and all the aggregated data of each, as set by User.

Type:

- Object

Properties:

Name	Type	Description
name	String	The name of the Dashboard as set by the User when created.
description	String	The brief description of the Dashboard pertaining to what is contained as set by the User when created.
is_public	Boolean	The flag to determine if the Dashboard should appear in the public Dashboard interface.
created_by	ObjectId	The ObjectId of the User responsible for creating the Dashboard.
blocks	Array.<ObjectId>	The array of ObjectId references to Block as set by the User when created.

Source: Documents/JSdocs/types.js, line 86

DataEntry

An object which contains all the AcquiSuite POST data as well as relevant relationships.

Type:

- Object

Properties:

Name	Type	Description
building	ObjectId	The ObjectId reference to Building that contains this DataEntry.
meter_id	ObjectId	The ObjectId reference to Meter that reported this DataEntry.
timestamp	String	The timestamp of the AcquiSuite POST data when it was collected.
point	Array.<Point>	The array of Point as reported by the AcquiSuite POST data.

Source: Documents/JSdocs/types.js, line 96

Google

An Object that contains details about a User's Google account

Type:

- Object

Properties:

Name	Type	Description
id	String	Contains the id of the Google acocunt.
token	String	Contains the auth token of the Google account.
email	String	Contains the email address of the Google account.
name	String	Contains the name of the user as it appears in their Google acocunt.

Source: Documents/JSdocs/types.js, line 57

Meter

An object which assists in determining where each DataEntry should go.

Type:

- Object

Properties:

Name	Type	Description
name	String	The name of the Meter. Typically contains an easily readable English string.
meter_id	String	The serial of the AcquiSuite with the address of the meter concatenated at the end.
building	ObjectId	The ObjectId reference to Building that the Meter connected to.

Source: Documents/JSdocs/types.js, line 105

`null`

Source: Documents/JSdocs/types.js, line 27
See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null

`Number`

Type:

- String | Number

Source: Documents/JSdocs/types.js, line 21
See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

`Object`

Source: Documents/JSdocs/types.js, line 11
See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

`ObjectId`

A 24-character hash string that references objects in a database. Assigned to each object entered into the database.

Type:

- String

Source: Documents/JSdocs/types.js, line 32
See: http://docs.mongodb.org/manual/reference/object-id/

Example

```
var id = user._id;

var id = new ObjectID(); // => such as '51b96d2a845cca7213000002'
```

`Point`

An Object that contains the formatted AcquiSuite POST data.

Type:

- Object

Properties:

Name	Type	Description
`number`	Number	Contains the index of itself in the containing array.
`name`	String	Contains the name of the value being measured.

Name	Type	Description
units	String	Contains the units of the value being measured.
value	Number	Contains the consumption value.

Source: Documents/JSdocs/types.js, line 48

Story

An object which contains Blocks as and all the aggregated data of each.

Type:

- Object

Properties:

Name	Type	Description
name	String	The name of the Story as set by the User when created.
is_public	Boolean	The flag to determine if the Story should appear in the public Story interface.
created_by	ObjectId	The ObjectId of the User responsible for creating the Story.
dashboards	Array.<ObjectId>	The array of ObjectId references to Dashboard as set by the User when created.

Source: Documents/JSdocs/types.js, line 113

String

Source: Documents/JSdocs/types.js, line 1
See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

User

An object which contains the login token and customizable settings of any given person using the interface.

Type:

- Object

Properties:

Name	Type	Description
google	Google	The token and email information of the User's Google account.
blocks	Array.<ObjectId>	The array of ObjectId references to Block that belong to this User.
dashboards	Array.<ObjectId>	The array of ObjectId references to Dashboard that belong to this User.
stories	Array.<ObjectId>	The array of ObjectId references to Story that belong to this User.

Name	Type	Description
accountAccess	String	The permission level of this account.

Source: Documents/JSdocs/types.js, line 122

Documentation generated by JSDoc 3.5.5 on Tue Jun 12 2018 15:29:28 GMT-0700 (Pacific Daylight Time) using the Minami theme.

7.4.2 Files

Source/app/models/block-schema.js

Contains the schema for our block object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/block-schema.js, line 1

Source/app/models/building-schema.js

Contains the schema for our building object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/building-schema.js, line 1

Source/app/models/dashboard-schema.js

Contains the schema for our dashboard object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/dashboard-schema.js, line 1

Source/app/models/data-entry-schema.js

Contains the schema for our dataentry object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/data-entry-schema.js, line 1

Source/app/models/meter-schema.js

Contains the schema for our meter object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/meter-schema.js, line 1

Source/app/models/story-schema.js

Contains the schema for our story object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/app/models/story-schema.js, line 1

Source/app/models/user-schema.js

Contains the schema for our user object.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/app/models/user-schema.js, line 1

Source/config/DBsampledata.js

Contains test functions to setup a dummy database.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/config/DBsampledata.js, line 1

Source/config/passport.js

Contains the functionality for our Google Passport.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/config/passport.js, line 1

Source/public/js/controllers/block-controller.js

Contains the controller for our block component.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/public/js/controllers/block-controller.js, line 1

Source/public/js/controllers/building-controller.js

Contains the controller for our building component.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/public/js/controllers/building-controller.js, line 1

Source/public/js/controllers/chart-controller.js

Contains the controller for our chart component.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/public/js/controllers/chart-controller.js, line 1

Source/public/js/controllers/dashboard-controller.js

Contains the controller for our dashboard component.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/public/js/controllers/dashboard-controller.js, line 1

Source/public/js/controllers/main-controller.js

Contains the controller for our main component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/controllers/main-controller.js, line 1

Source/public/js/controllers/map-controller.js

Contains the controller for our map component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/controllers/map-controller.js, line 1

Source/public/js/controllers/meter-controller.js

Contains the controller for our meter component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/controllers/meter-controller.js, line 1

Source/public/js/controllers/side-nav-controller.js

Contains the controller for our side navigation bar component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/controllers/side-nav-controller.js, line 1

Source/public/js/controllers/story-controller.js

Contains the controller for our story component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/controllers/story-controller.js, line 1

Source/public/js/core.js

Contains the redirect routing for our website. Also contains an edit controller and user controller

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/core.js, line 1

Source/public/js/services/block-service.js

Contains the services for our block component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/block-service.js, line 1

Source/public/js/services/building-service.js

Contains the services for our building component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/building-service.js, line 1

Source/public/js/services/chart-service.js

Contains the services for our chart component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/chart-service.js, line 1

Source/public/js/services/dashboard-service.js

Contains the services for our dashboard component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/dashboard-service.js, line 1

Source/public/js/services/meter-service.js

Contains the services for our meter component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/meter-service.js, line 1

Source/public/js/services/story-service.js

Contains the services for our story component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/story-service.js, line 1

Source/public/js/services/user-service.js

Contains the services for our user component.

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/public/js/services/user-service.js, line 1

Source/server.js

Contains the functions for server setup

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/server.js, line 1

Documentation generated by JSDoc 3.5.5 on Tue Jun 12 2018 15:29:29 GMT-0700 (Pacific Daylight Time) using the Minami theme.

7.4.3 DataServer

DataServer

Contains the functions for data collection server backend

Author:	Aubrey Thenell, Daniel Schroede, Parker Bruni.
Source:	Source/dataServer.js, line 1

Methods

`(inner) addEntry(meter, body, serialAddress) → {Promise}`

Function to add a data entry to a database. Checks for duplicate first. Conditions: - Called whenever POST request comes in

Parameters:

Name	Type	Description
`meter`	Object	Contains elements of the source meter *Properties* **Name** / **Type** / **Description** `_id` — String — Contains the ID of the meter as it is in the database `name` — String — Contains the public facing name of the meter as it is in the database `building` — String — Contains the building that the meter is associated with as it is in the database
`body`	Object	Contains the POST request body with all extraneous pieces cut
`serialAddress`	Object	Contains the POST request body that can access the serial and address information

Source:	Source/dataServer.js, line 310

Returns:

Type `Promise`

`(inner) addMeter(meter) → {Promise.<Meter>}`

Function that is called when the server gets a POST request from a meter that is not in our database already. Only called when a matching meter cannot be located in the database

Parameters:

Name	Type	Description
meter	Object	Contains a POST request body that is slightly trimmed

Source: Source/dataServer.js, line 283

Returns:

newmeter

Type Promise.<Meter>

(inner) checkMeterTimestamps() → {void}

Helper function to see when the last time a meter posted Called once per day and Sends email if meter hasn't checked in for over a day but under two

Source: Source/dataServer.js, line 172

Fires:

- event:emailAlert

Returns:

Type void

(inner) checkUsage()

Function used to find and alert during high usage spikes Conditions - Only sends email if Average for day is greater than the average for week plus a standard deviation and the meter has more than 100 entries

Source: Source/dataServer.js, line 214

Fires:

- event:emailAlert

(inner) emailAlert(email) → {void}

Sends out alert to users (admins). Specifically when A meter hasn't checked in over a day. Also when a meter reports unusually high usage.

Parameters:

Name	Type	Description

Name	Type	Description
email	Object	Contains email contents to be sent. *Properties* **Name** / **Type** / **Description** subject / String / Contains the email's subject. body / String / Contains the email's body.

Source: Source/dataServer.js, line 119

Returns:

Type `void`

`(inner) post->receiveXML(path, xml) → {Promise}`

Handles POST requests received in XML format Receives POST requests, converts from XML to JSON

Parameters:

Name	Type	Description
path	string	Express path
xml	Object	Contains the XML data coming in.

Source: Source/dataServer.js, line 48

Fires:

- event:checkUsage
- event:checkMeterTimestamps
- event:addMeter
- event:addEntry

Returns:

Type `Promise`

7.4.4 PublicServer

PublicServer

Contains the Express routes for the server responsible for user input as well as serving content.

Author: Aubrey Thenell, Daniel Schroede, Parker Bruni.

Source: Source/app/routes.js, line 1

Methods

`(inner) addMeter(meter, savedBuilding) → {Promise}`

Function for adding Meters to a building.

Parameters:

Name	Type	Description
`meter`	String	id of Meter being added.
`savedBuilding`	String	id of Building being added to.

Source: Source/app/routes.js, line 1674

Fires:

- event:pushNullMeter

Returns:

Type `Promise`

`(inner) Block/get->api/deleteBlock(path, callback) → {Block}`

Route for retrieving a block given an id.

Parameters:

Name	Type	Description
`path`	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains request data. *Properties* **Name** **Type** **Description** query Object Contains passed query. *Properties* **Name** **Type** **Description** block_id String The id of the block being retrieved.

Source: Source/app/routes.js, line 614

Returns:

Type Block

`(inner) Block/get->api/getBlocksForDashboards(path, callback) → {Array.<Block>}`

Route for returning an array of Blocks to populate a Dashboard

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains request data. *Properties* **Name** **Type** **Description** user User contains user to retrieve the blocks of.

Source: Source/app/routes.js, line 500

Returns:

Type Array.<Block>

(inner) Block/get->api/getUserBlocks(path, callback) → {Array.<Blocks>}

Route for returning a given users blocks.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* Name: req.user, Type: User, Description: contains user to retrieve the blocks of.

Source: Source/app/routes.js, line 473

Returns:

Type Array.<Blocks>

(inner) Block/post->api/deleteBlock(path, callback) → {String}

Route for deleting a block from the database and the user's assosciation

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains request data. *Properties* **Name** / **Type** / **Description** user / User / contains user to delete block from body / Block / contains Block to delete

Source: Source/app/routes.js, line 577

Returns:

Type String

(inner) Block/post->api/getBlocksForDashboards(path, callback) → {Array.<Block>}

Route for returning all of the data entries assosciated with a building, given an id

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains request data. *Properties* **Name** / **Type** / **Description** user / User / contains user to add Block to.

Source: Source/app/routes.js, line 529

Returns:

Type Array.<Block>

(inner) Block/post->api/updateBlock(path, callback) → {Block}

Route for updating a block given an id.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains request data. *Properties* **Name** **Type** **Description** body Object Contains passed query. *Properties* **Name** **Type** **Description** name String The new name of the block. chart Chart The new chart data. building String The id of the assosciated building. is_public Boolean The bool to determine if the public can view it. variable String The unit being measured.

Source: Source/app/routes.js, line 640

Returns:

Type Block

(inner) Building/get->api/buildings(path, callback) → {Array.<Building>}

Route for returning all of the buildings stored in the database to the application.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 164

Returns:

Type Array.<Building>

(inner) Building/get->api/getBuildingById(path, callback) → {Building}

Route for returning a building given a MongoDB id tag.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 185

Returns:

Type Building

(inner) Building/get->api/getBuildingData(path, callback) → {Object}

Route for returning all of the data entries assosciated with a building, given an id

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains request data. *Properties* **Name** **Type** **Description** query Object Contains passed query. *Properties* **Name** **Type** **Description** buildings Array.<String> Array of building ids. start String Start time to filter. end String End time to filter. var String The unit type.

Source: Source/app/routes.js, line 209

Returns:

Type Object

(inner) Building/get->singleLine(path, callback) → {FILE}

Route for returning a single line and returns the html file.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 446

Returns:

Type FILE

(inner) Building/post->api/addBuilding(path, callback) → {Building}

Route for adding buildings through the application interface.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 55

Fires:

- event:updateOldBuildingMeters
- event:addMeter

Returns:

Type Building

(inner) Building/post->api/deleteBuilding(path, callback) → {String}

Route for deleting buildings through the application interface.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 126

Returns:

Type String

(inner) Building/post->api/toCSV() → {FILE}

Route for exporting building data to a csv file.

Parameters:

Name	Type	Description
callback.req.query.buildings	Array.<String>	Array of building ids.

Name	Type	Description
`callback.req.query.start`	String	Start time to filter.
`callback.req.query.end`	String	End time to filter.

Source: Source/app/routes.js, line 1349

Returns:

Type `FILE`

`(inner) Building/post->api/updateBuilding(path, callback) → {Building}`

Route for updating buildings through the application interface.

Parameters:

Name	Type	Description
`path`	string	Express path.
`callback`	callback	Express middlewear.

Source: Source/app/routes.js, line 86

Fires:

- event:updateOldBuildingMeters
- event:addMeter

Returns:

Type `Building`

`(inner) Dashboard/get->api/getDashboardNames(path, callback) → {Array.<Dashboard>}`

Route for getting the names of a given User's Dashboards.

Parameters:

Name	Type	Description
`path`	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** user User contains user to retrieve the Dashboards of.

Source: Source/app/routes.js, line 794

Returns:

Type Array.<Dashboard>

(inner) Dashboard/get->api/getDashboards(path, callback) → {Array.<Dashboard>}

Route for returning a given users dashboards.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** user User contains user to retrieve the Dashboards of.

Source: Source/app/routes.js, line 731

Returns:

Type Array.<Dashboard>

(inner) Dashboard/get->api/getPublicDashboards(path, callback) → {Array.<Dashboard>}

Route for returning the Dashboards labeled public.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** user User contains user to retrieve the Dashboards of.

Source: Source/app/routes.js, line 764

Returns:

Type Array.<Dashboard>

(inner) Dashboard/post->api/addDashboard(path, callback) → {User}

Route for adding a new dashboard from the application interface.

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains request data. *Properties* **Name** **Type** **Description** body Object Contains passed query. *Properties* **Name** **Type** **Description** name String The name of the new dashboard. description String The description of the new dashboard. user String The user who created the dashboard. is_public Boolean The bool that determines if the public can see it or not. blocks Array.<Block> The array of Blocks to add to Dashboard.

Source: Source/app/routes.js, line 681

Returns:

Type User

`(inner) Dashboard/post->api/deleteDashboard(path, callback) → {String}`

Route for deleting a Dashboard given a user id.

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** user User contains user to delete the Dashboard from. body Dashboard contains the Dashboard to delete.

Source: Source/app/routes.js, line 822

Returns:

Type String

(inner) Dashboard/post->api/updateDashboard(path, callback) → {Dashboard}

Route for updating a Dashboard given the Dashboard.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** body Dashboard Contains Dashboard.

Source: Source/app/routes.js, line 858

Returns:

Type Dashboard

(inner) Email/post->api/emailRegistration(path, callback) → {String}

Route for emailing and updating user credentials based on query.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** body Object Contains an object with email and user info. *Properties* **Name** **Type** **Description** email String Contains the email address. access String Contains the User's new account access level.

Source: Source/app/routes.js, line 1214

Returns:

Type String

(inner) getMcNaryDining(match) → {Promise}

Function for adding retrieving McNary Dining specifically because it has special properties.

Parameters:

Name	Type	Description
match	Object	timestamp to crosscheck McNary entries..

Source: Source/app/routes.js, line 1708

Fires:

- event:pushNullMeter

Returns:

Type `Promise`

`(inner) Google->api/google_user)(path, callback) → {google}`

Route serving login authorization.

Parameters:

Name	Type	Description
`path`	string	Express path.
`callback`	callback	Express middlewear.

Source: Source/app/routes.js, line 36

Returns:

Type `google`

`(inner) Google/get->auth/google() → {callback}`

Route for Google Authentication. Send to google to do the authentication. Profile gets us their basic information including their name. Email gets their emails.

Source: Source/app/routes.js, line 1302

Fires:

- event:Google/get->auth/google/callback

Returns:

Type `callback`

`(inner) Google/get->auth/google() → {FILE}`

Route for Google Authentication, the callback after google has authenticated the user. Redirects User on success/failure.

Source: Source/app/routes.js, line 1319

Returns:

Type `FILE`

(inner) Google/get->logout() → {FILE}

Route for logging a User out of their Google account.

Source: Source/app/routes.js, line 1335

Returns:

Type FILE

(inner) isLoggedIn() → {callback}

Route middleware to make sure a user is logged in

Source: Source/app/routes.js, line 1862

Returns:

Type callback

(inner) login/get->login(path, callback) → {FILE}

Route for returning and rendering the login splash html page.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 459

Returns:

Type FILE

(inner) Meter/get->api/getMeterById(path, callback) → {Meter}

Route for retrieving a Meter given an id.

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains query. *Properties* **Name** / **Type** / **Description** query.meter_id / String / Contains Meter being retrieved..

Source: Source/app/routes.js, line 1126

Returns:

Type Meter

(inner) Meter/get->api/getMeters(path, callback) → {Array.<Meter>}

Route for retrieving Meters.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 1104

Returns:

Type Array.<Meter>

(inner) Meter/post->api/addMeter(path, callback) → {Meter}

Route for adding a Meter.

Parameters:

Name	Type	Description
path	string	Express path.

Name	Type	Description
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** user User Contains the User who added the Meter. body Meter Contains the Meter to add.

Source: Source/app/routes.js, line 1077

Returns:

Type Meter

`(inner) Meter/post->api/addMeter(path, callback) → {Meter}`

Route for deleting a Meter.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** body Meter Contains the Meter to delete.

Source: Source/app/routes.js, line 1184

Returns:

Type Meter

(inner) Meter/post->api/updateMeter(path, callback) → {Meter}

Route for updating a Meter.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains query. *Properties* **Name** / **Type** / **Description** body / Meter / Contains the Meter to update.

Source: Source/app/routes.js, line 1151

Returns:

Type Meter

(inner) pushNullMeter(meter, savedBuilding) → {Promise}

Function that adds DataEntries to a Building when the Meter is first added to that Building.

Parameters:

Name	Type	Description
meter	String	id of Meter being added.
savedBuilding	String	id of Building being added to.

Source: Source/app/routes.js, line 1800

Returns:

Type Promise

(inner) Story/get->api/getPublicStories(path, callback) → {Array.<Story>}

Route for retrieving public Stories.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains query.

Source: Source/app/routes.js, line 1043

Returns:

Type Array.<Story>

(inner) Story/get->api/getUserStories(path, callback) → {Array.<Story>}

Route for getting stories given a User id.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** / **Type** / **Description** req / Object / Contains query. *Properties* **Name** / **Type** / **Description** user / User / Contains User to retrieve stories..

Source: Source/app/routes.js, line 893

Returns:

Type Array.<Story>

(inner) Story/get->storyNav(path, callback) → {FILE}

Route for returning and rendering the story selection html page.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear.

Source: Source/app/routes.js, line 432

Returns:

Type FILE

(inner) Story/post->api/addStory(path, callback) → {User}

Route for adding a Story to a given User.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* Name / Type / Description req — Object — Contains query. *Properties* Name / Type / Description user — User — Contains User. body — Story — Contains Story to add to user.

Source: Source/app/routes.js, line 928

Returns:

Type User

(inner) Story/post->api/deleteStory(path, callback) → {Story}

Route for deleting a Story.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** body Story Contains Story to delete

Source: Source/app/routes.js, line 1008

Returns:

Type Story

`(inner) Story/post->api/updateStory(path, callback) → {Story}`

Route for updating a Story.

Parameters:

Name	Type	Description
path	string	Express path.
callback	callback	Express middlewear. *Properties* **Name** **Type** **Description** req Object Contains query. *Properties* **Name** **Type** **Description** body Story Contains Story to add to user

Source: Source/app/routes.js, line 974

Returns:

Type Story

(inner) updateOldBuildingMeters(meter, building) → {PROMISE}

Function to update a Building's Meter's when that specific Building is Edited. Also called when new Building is created with in use meters.

Parameters:

Name	Type	Description
meter	Array.<String>	Array of Meter IDs
building	Building	Building to check and remove meters from

Source: Source/app/routes.js, line 1637

Returns:

Type PROMISE

---

8 Recommended Technical Resources for Learning More

8.1 Helpful Websites(Listed in order of helpfulness)

- Github
- Slack
- Stack Overflow
- W3 Schools

8.2 Helpful People on Campus

- Kirsten Winters
- Kevin McGrath
- Behnam Saeedi
- Jack Woods

9 Conclusions and Reflections

9.1 Daniel

9.1.1 What technical information did you learn?

Before building this project, I had very little (if any) knowledge about AngularJS as a front-end framework, MongoDB, Amazon web services, or Express.js. While I had previously tried to do small scale M.E.A.N. stack applications, they were always bootstrapped from an example code source and I never truly understood the fundamentals behind a RESTful API or a full stack web application. I think building this project allowed me to really learn 3 main concepts:

- Read and learn from documentation
- Reach proficiency with the M.E.A.N. stack frameworks
- Understand how to design and implement a large software architecture

This was my first project built from scratch of something to this size and scale and it taught me a lot about the design process. I now understand why it is so common for companies to migrate legacy monolith architectures into smaller microservices. Systems that are extremely large become easily cluttered and become filled with minute work-arounds or case-dependent implementations that are not good for readability or robustness. I found that we lacked a large amount of understanding during the design process and it made a huge impact during the implementation of our project.

9.1.2 What non-technical information did you learn?

The biggest non-technical skill I learned during the course of this project was time-management and team-management. Towards the end of the project, I began setting myself hard deadlines for feature completion and implementation which proved to be extremely helpful for finishing everything that needed to get done. During the winter term, I would pick away at something here, do a little bit on something there, but never set myself deadlines for finishing up concrete requirements. I found myself becoming more and more relaxed with finishing up code and that made me fall behind at the start of spring term. I think better planning with both the architecture design and Gantt chart/read map would have proven extremely helpful throughout the development of this project.

9.1.3 What have you learned about project work/project management?

Planning and sharing work to ensure completion is crucial. Towards the end of the project, I found myself assigning personal tasks and tasks to group members with hard deadlines to speed up completion. Prior to having concrete due dates for features, the rate of implementation was slow and it seemed we were always crunched for time. I think breaking the project up into small pieces and following a concrete road map for implementations would have proven extremely helpful.

9.1.4 What have you learned about working in teams?

I think it is important to lay out your values and expectations of yourself and others at the beginning of the project. I found that my expectations for quality and time-management differed from my group members. Putting this on the table and discussing what everyone expects from each other I think would have reduced a lot of that uncertainty and frustration throughout the year.

9.1.5 If you could do it all over, what would you do differently?

There are specific technical changes I would make to this project if I could do it all over again. First and foremost, for the amount of relations we had in our data, I think using MySQL as a back-end would have benefited us immensely. Secondly, and as mentioned above, I think that a thorough Gantt chart and implementation roadmap would have assisted the flow and overall quality of the project. If we had a lot of the necessary backend requirements or charting implementations done sooner, it would have alleviated a lot of the debugging and error testing that was needed towards the end of the project. I would have also like to have everyone working on and understanding every aspect of the project. Learning curves and skill-level limitations played a large role in preventing contributions in many areas of the project.

9.2 Parker

9.2.1 What technical information did you learn?

I learned a huge amount of new information on web development technologies that I had previously had no knowledge of. I learned the ins and outs of what it takes to develop a full stack web application, and while most of my efforts were on UI, I learned a decent bit about backend frameworks as well as chart.js and google maps implementations.

9.2.2 What non-technical information did you learn?

My teamwork skills have greatly improved. Specifically, communication within a team environment and the ability to adapt to different work styles to better suit the needs of the team. As well, I feel that my work ethic for large scale projects has improved. Specifically, the ability to assess requirements and the amount of work needed as well as estimating when and how much to work in order to meet deadlines.

9.2.3 What have you learned about project work?

I know now how crucial it is to divide the work into manageable pieces between people and establish good communication between those working parties so that the project can come together as a whole in a functional format. I have previously not worked on a project of this scale for this amount of time so I have gained some good insights on how the work on a project should be handled.

9.2.4 What have you learned about project management?

I learned that a project works well when someone on the team takes the lead role, namely Daniel. This seemed to promote work ethic within the group and helped us stay organized to meet deadlines for assignments and figure out who needs to do what portions of the project and when.

9.2.5 What have you learned about working in teams?

Teamwork flows much better with good communication. The web based communication application ”Slack” seemed to let us work much more efficiently as it allowed for instant text communication and file sharing. This workspace is organized with team collaboration in mind and I would recommend it to any team that is working on large scale projects. Communication is key, but so is group members who are willing and open to new ideas about changes or other aspects of the projects. In person team meetings were good to establish good relationships and in person interactions to get work done.

9.2.6 If you could do it all over, what would you do differently?

I would ideally know more about web based technologies, as I had a relatively small amount of knowledge of potential technologies and if I had known more potential technologies and their pros and cons we would have been able to make decisions about the project much quicker and it is likely that the final product would function more efficiently.

9.3 Aubrey

9.3.1 What technical information did you learn?

This project has really solidified my understanding of AngularJS and related frameworks. Between this project and the projects I'm assigned at work, I am fairly solidified in what I'd like to focus on once I graduate. The most important technical takeaway is my backend knowledge. Since I focused building the backend side of the project, I have gained quite a bit of experience. This makes me happy because backend knowledge is applicable everywhere. Even if the language or framework is different, the concepts will be the same.

9.3.2 What non-technical information did you learn?

I learned that my time management has some room for improvement. Between this project, working over 20 hours a week, and taking 19 credit hours, I feel that my time management and work ethic have improved drastically over the course of the project. Another non-technical thing I learned was about energy. Through talks with Jack and the Office of Sustainability, my understanding of how sustainable systems work as well as how Oregon State University implements techniques to make itself more sustainable.

9.3.3 What have you learned about project work/management?

I think my main takeaway from this section would be to not underestimate how much time a task will take. A seemingly simple task can (and has) take dozens of hours when I only thought a few hours were needed. I also think that breaking down tasks into smaller subtasks will not only make it seem like less, but will also make it less daunting.

9.3.4 What have you learned about working in teams?

This has definitely been the most stressful group project I've done. While we got along just fine, it was rather difficult to get everyone on the same page on what needed to be done and how long dependency tasks were going to take. I think if we understood the scope of the project better from the get go, we could have worked together and figured out a better work distribution that would prevent less constant progress blocking.

9.3.5 If you could do it all over, what would you do differently?

First and foremost I would choose the more modern Angular2+ over AngularJS. Combine this with typescript and I feel like this would have helped us keep things better modularized. I suggested Angular2+ during the design phase, but we decided to go with AngularJS because Daniel had a decent amount of experience already. I would also want to try out SQL to see how performances differed. There were times when I was writing mongo queries that just made me wish I could use SQL queries instead. Granted, Mongo does support SQL queries, but the performance isn't as good as native SQL, so not much point using them. I would also start on the coding aspect of the project sooner. If we had a lot of the information about how the AcquiSuites worked from the get go, that would have saved me several dozen hours of debugging.

10 Appendix 1: Essential Code Listings.

10.0.1 Node.js crypto salt/hash password example

```
'use strict';
var crypto = require('crypto');
/**
    * generates random string of characters i.e salt
    */
var genRandomString = function(length){
    return crypto.randomBytes(Math.ceil(length/2))
            .toString('hex') /** convert to hexadecimal format */
            .slice(0,length);   /** return required number of characters */
};
/** hash password with sha512.
    * @function
    * @param {string} password - List of required fields.
    * @param {string} salt - Data to be validated.
    */
var sha512 = function(password, salt){
    var hash = crypto.createHmac('sha512', salt); /** Hashing algorithm sha512 */
    hash.update(password);
    var value = hash.digest('hex');
    return {
        salt:salt,
        passwordHash:value
    };
};
function saltHashPassword(userpassword) {
    var salt = genRandomString(16); /** Gives us salt of length 16 */
    var passwordData = sha512(userpassword, salt);
    console.log('UserPassword = '+userpassword);
    console.log('Passwordhash = '+passwordData.passwordHash);
    console.log('nSalt = '+passwordData.salt);
}
```

Listing 1: An example of how to salt hash passwords using the Node.js crypto module (Taken from *Rahil Shaikh's example*) [38]

10.0.2 Using Passport.js to include Google oAuth 2.0 authentication

```
var passport = require('passport');
var GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;

// Use the GoogleStrategy within Passport.
//   Strategies in Passport require a `verify` function, which accept
//   credentials (in this case, an accessToken, refreshToken, and Google
//   profile), and invoke a callback with a user object.
passport.use(new GoogleStrategy({
    clientID: GOOGLE_CLIENT_ID,
    clientSecret: GOOGLE_CLIENT_SECRET,
    callbackURL: "http://www.example.com/auth/google/callback"
    },
    function(accessToken, refreshToken, profile, done) {
        User.findOrCreate({ googleId: profile.id }, function (err, user) {
            return done(err, user);
        });
    }
));
```

Listing 2: Using passport to include Google oAuth 2.0 authentication. (Taken from *Passport Documentation*) [39]

10.0.3 Using Passport.js to maintain user sessions.

```
passport.serializeUser(function(user, done) {
    done(null, user.id);
    });

    passport.deserializeUser(function(id, done) {
    User.findById(id, function(err, user) {
        done(err, user);
    });
    });
```

Listing 3: The user ID is serialized to the session, keeping the amount of data stored within the session small. When subsequent requests are received, this ID is used to find the user, which will be restored to req.user. (Taken from *Passport Documentation*) [21]

10.0.4 Using UI Bootstrap to create a datepicker object.

```
    // HTML Declaration
    <input type="text" class="form-control"
        datepicker-popup="{{format}}"
        ng-model="dt"
        is-open="opened"
        min-date="minDate"
        max-date="'2015-06-22'"
        datepicker-options="dateOptions"
        date-disabled="disabled(date, mode)"
        ng-required="true"
        close-text="Close"
        id="date-picker"
        readonly
        ng-click="open($event)"
    />

    // Angular Controller
    var calPicker = angular.module("calPicker", ['ui.bootstrap']);

    calPicker.controller("DatepickerDemoCtrl", ["$scope", function($scope){

        // grab today and inject into field
        $scope.today = function() {
        $scope.dt = new Date();
        };

        // run today() function
        $scope.today();

        // setup clear
        $scope.clear = function () {
        $scope.dt = null;
        };

        // open min-cal
        $scope.open = function($event) {
        $event.preventDefault();
        $event.stopPropagation();

        $scope.opened = true;
        };

        // handle formats
        $scope.formats = ['dd-MMMM-yyyy', 'yyyy/MM/dd', 'dd.MM.yyyy', 'shortDate'];

        // assign custom format
        $scope.format = $scope.formats[0];

    }]);
```

Listing 4: An example of using UI Bootstrap to create a datepicker object with an Angular controller. (Taken from *Angular - Bootstrap UI - Datepicker*) [40]

10.0.5 XML Data Acquisition

```
    app.post('/receiveXML', xmlparser({
    trim: false,
    explicitArray: false
}), function (req, res) {
    if (req.body.das.mode === 'LOGFILEUPLOAD') {
        timestamp = moment().utc().format('HH:mm:ss');
        console.log('Received XML data on: ' + moment.utc().format('YYYY-MM-DD HH:mm:ss'));

        // calls email helper func once per day.
        if (emailFlag && timestamp > '00:00:00' && timestamp < '00:20:00') {
            console.log('Meters being reviewed for outage.')
            emailFlag = false;
            dataFlag = false;
            checkUsage();
            checkMeterTimestamps();
        } else if (!emailFlag && timestamp > '12:00:00' && timestamp < '12:15:00') {
            console.log('Email flag being reset');
```

```
            emailFlag = true;
            dataFlag = true;
        }
        pathShortener = req.body.das.devices.device.records;

        // Checks if meter exists. If it doesn't adds one.
        // Then/else adds incoming data entry
        Meter.findOne({
            meter_id: (req.body.das.serial + '_' + req.body.das.devices.device.address)
        }, (err, doc) => {
            if (!doc) {
                addMeter(req.body.das).then(data => addEntry(data, pathShortener, req.body.das));
            } else {
                addEntry(doc, pathShortener, req.body.das);
            }
        });
    } else {
        console.log('STATUS file received');
    }
    res.status("200");
    res.set({
        'content-type': 'text/xml',
        'Connection': 'close'
    });
    res.send("<?xml version='1.0' encoding='UTF-8' ?>\n" +
        "<result>SUCCESS</result>\n" +
        "<DAS></DAS>" +
        "</xml>");
});
```

Listing 5: The function AcquiSuites send data to.

10.0.6 Add Data Entry

```
    function addEntry(meter, body, serialAddress) {
    console.log(body);
    return new Promise((resolve, reject) => {
        entryArray = new Array();
        if (body.record.length == undefined) {
            console.log('-------------------------- XML DATA --------------------------');
            console.log(`AcquiSuite: ${serialAddress.serial}\tAddress: ${serialAddress.devices.device
                .address}`);
            console.log(`Timestamp: ${pathShortener.record.time._}`);
            console.log(`point[0]: ${pathShortener.record.point[0].$.value}`);
            console.log('-------------------------------------------------------------');
            entry = new DataEntry();
            entry.meter_id = meter._id;
            entry.timestamp = body.record.time._;
            entry.building = meter.building;
            body.record.point.forEach((e, i) => {
                entry.point[i] = e.$;
                entry.point[i].value = Math.abs(entry.point[i].value);
            });
            entryArray.push(entry);
        } else {
            console.log('-------------------------- XML DATA --------------------------');
            console.log(`AcquiSuite: ${serialAddress.serial}\tAddress: ${serialAddress.devices.device
                .address}`);
            console.log(`Timestamp: ${pathShortener.record[0].time._}`);
            console.log(`point[0]: ${pathShortener.record[0].point[0].$.value}`);
            console.log('-------------------------------------------------------------');
            for (var i = 0; i < body.record.length; i++) {
                entry = new DataEntry();
                entry.meter_id = meter._id;
                entry.timestamp = body.record[i].time._;
                entry.building = meter.building;
                body.record[i].point.forEach((e, i) => {
                    entry.point[i] = e.$;
                    entry.point[i].value = Math.abs(entry.point[i].value);
                });
                entryArray.push(entry);
            }
        }
        entryArray.forEach(x => {
            DataEntry.findOne({
```

```
                timestamp: x.timestamp,
                meter_id: meter._id
            }, (err, doc) => {
                if (doc === null || doc === undefined) {
                    // save it to data entries
                    x.save().catch(err => {
                        res.status(400)
                    })
                    // add it to building
                    if (x.building != null && x.building != 'null') {
                        Building.findOneAndUpdate({
                                _id: entry.building
                            }, {
                                $push: {
                                    data_entries: x
                                }
                            },
                            (err) => {
                                if (err) throw (err)
                            });
                    }
                    console.log('Data entry id "' + x._id + '" with timestamp ' + x.timestamp + '
                        added to the meter named "' + meter.name + '" which is assigned to building
                        id: "' + meter.building + '"');
                } else {
                    console.log('Duplicate detected and nothing has been added!');
                    console.log('Incoming Data\'s timestamp:\t' + x.timestamp + '  meter_id:\t' + x.
                        meter_id);
                    console.log('Existing Data\'s timestamp:\t' + doc.timestamp + '  meter_id:\t' +
                        doc.meter_id);
                }
            });
        });
        resolve()
    });
}
```

Listing 6: The function that is called to add incoming data entries to the database.

11 APPENDIX 2: ANYTHING ELSE YOU WANT TO INCLUDE.

Photos, etc.

11.1 Design Document Images

11.1.1 Home Webpage (Public Access)

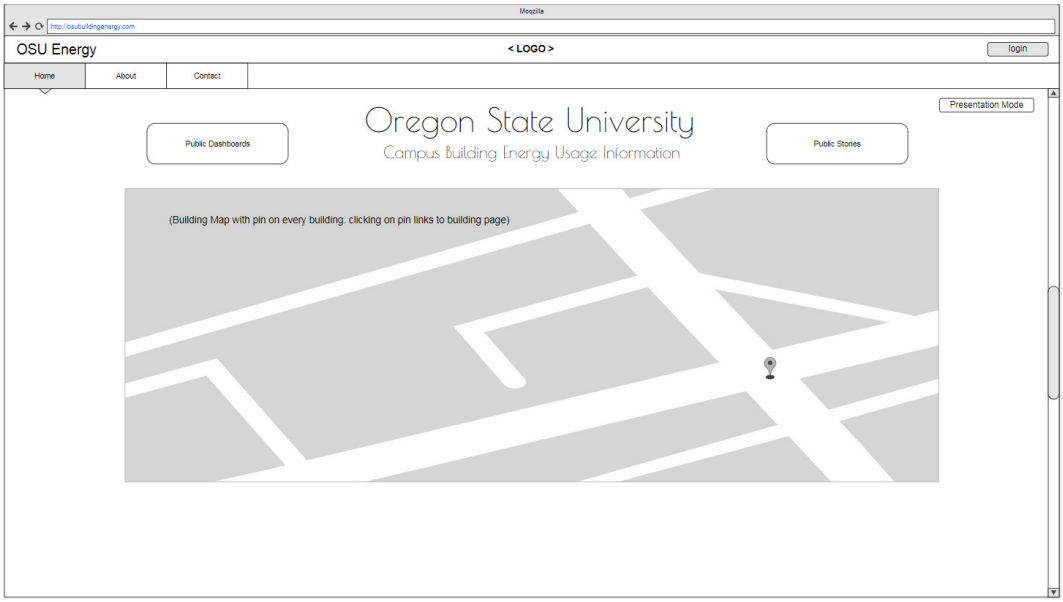


Figure 12: A mock-up of a home page from a general public user access perspective (not logged in to an account).

11.1.2 Home Webpage (Logged in Access)

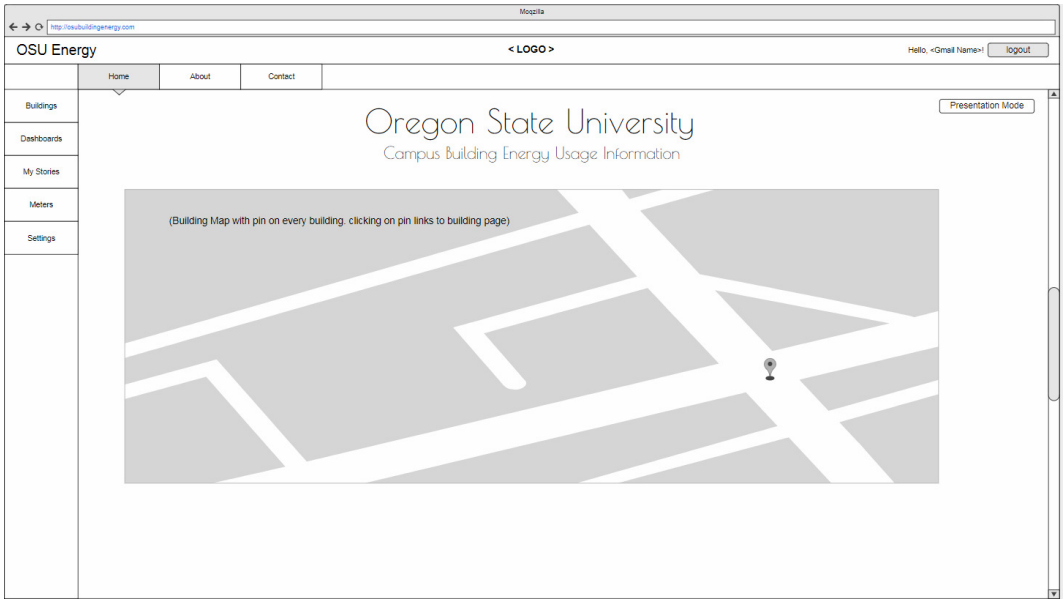


Figure 13: A mock-up of a home page from an authorized user access perspective (logged in to an account).

11.1.3 Buildings Webpage

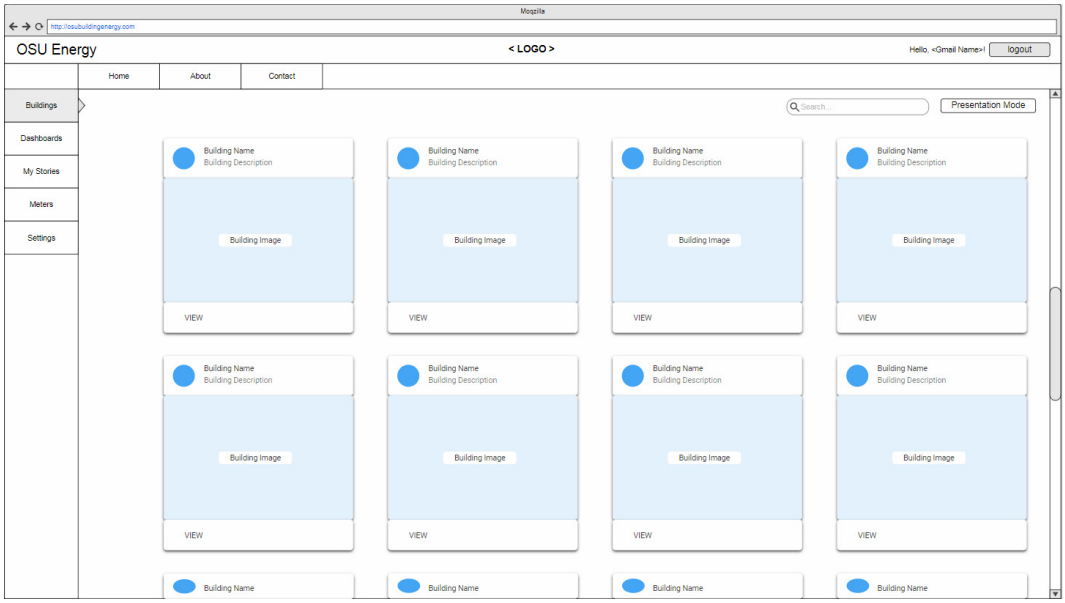


Figure 14: A mock-up of the webpage that shows all buildings.

11.1.4 Selected Building Page

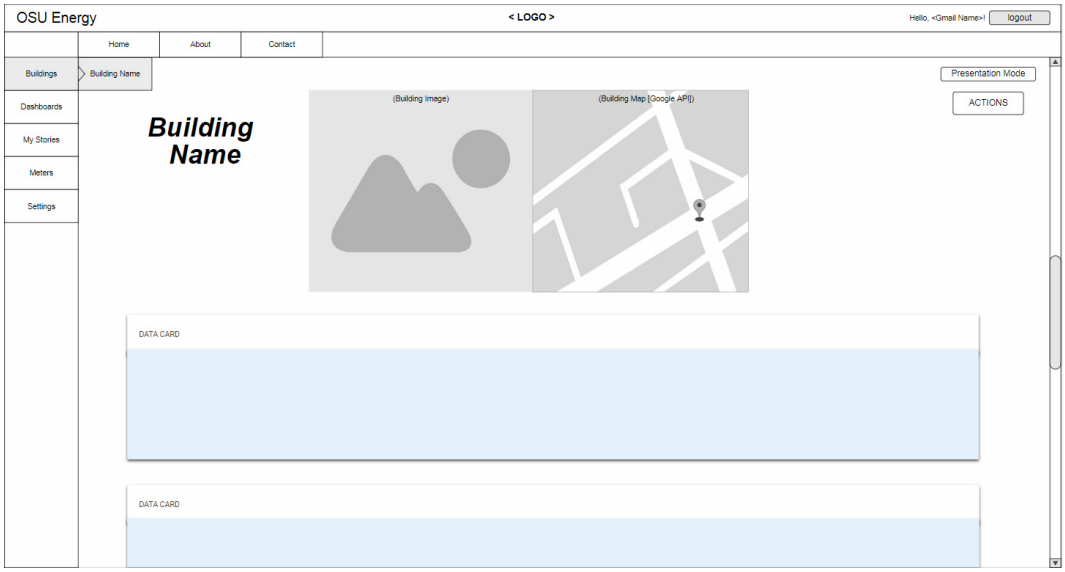


Figure 15: A mock-up of a selected building page.

11.1.5 Dashboards Page

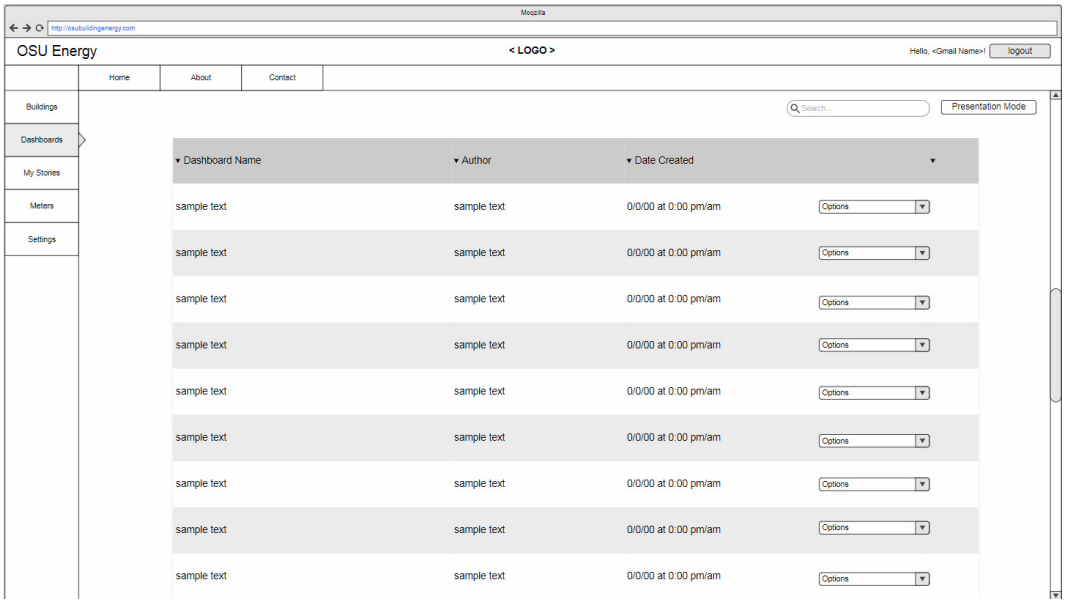


Figure 16: A mock-up page of a list of user generated dashboards.

11.1.6 Selected Dashboard Webpage

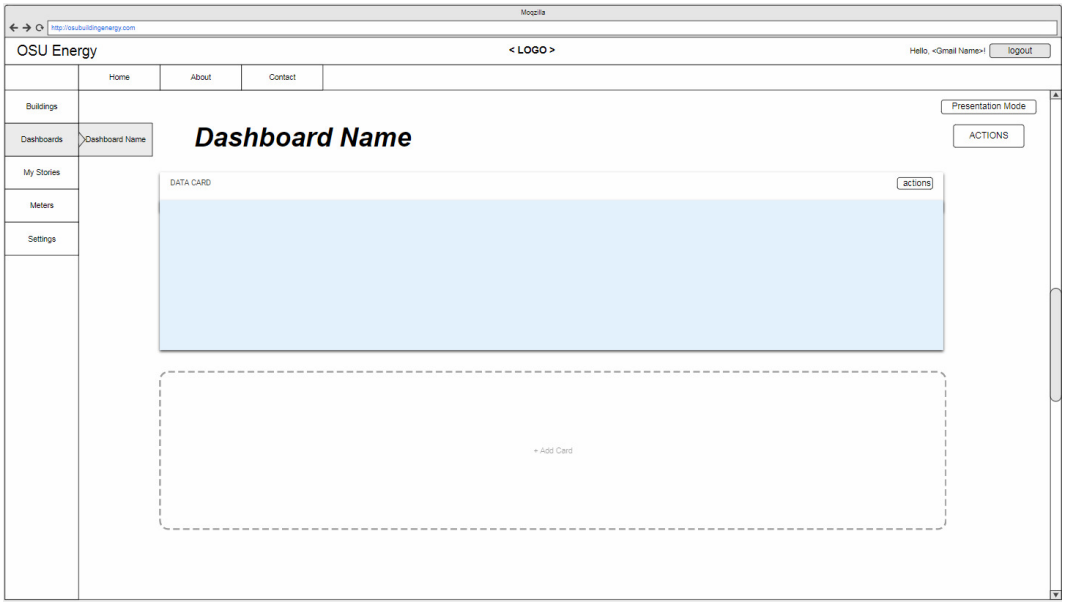


Figure 17: A mock-up of a selected dashboard webpage.

11.1.7 Stories Webpage

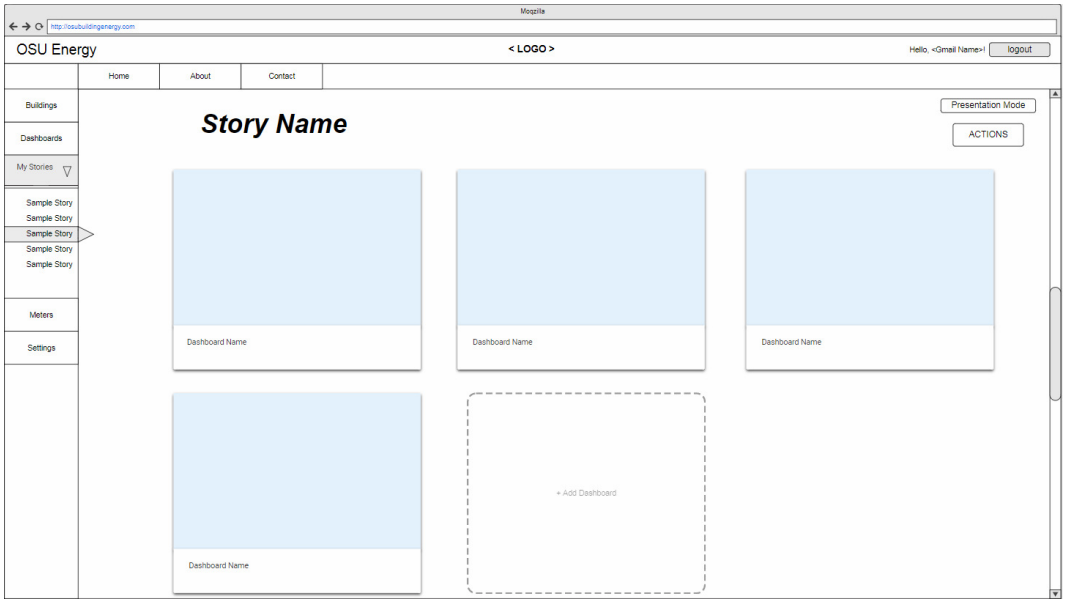


Figure 18: A mock-up of a list of user generated story Webpages.

REFERENCES

[1] AcquiSuite. (2017) Acquisuite emb. [Online]. Available: http://www.obvius.com/Products/A8810-0/
[2] obviusenergy. (2017) Modbus addressing & wiring best practices. [Online]. Available: https://www.youtube.com/watch?v=7x0tQf1OsDY
[3] L. Engineering. (2017) Modbus how does modbus work modbus tutorial. [Online]. Available: https://www.youtube.com/watch?v=jfKY928Ozrw/
[4] obvius.com. (2017) Acquisuite. [Online]. Available: http://www.obvius.com/Products/A8812
[5] D3.js. D3 data driven documents. [Online]. Available: https://d3js.org/
[6] Vis.js. vis.js. [Online]. Available: http://visjs.org/
[7] MEAN.IO. (2017) Mongodb, express, angularjs, node.js. [Online]. Available: http://mean.io
[8] JSON. (2017) Introducing json. [Online]. Available: http://json.org
[9] BuildingOS. (2017) Buildingos. [Online]. Available: https://buildingos.com/
[10] Mochajs. (2017) Mocha. [Online]. Available: https://mochajs.org/
[11] v4 alpha.getbootstrap.com. (2017) Dashboard. [Online]. Available: https://v4-alpha.getbootstrap.com/examples/dashboard/
[12] Google. Using oauth 2.0 to access google apis. [Online]. Available: https://developers.google.com/identity/protocols/OAuth2
[13] d3js.org. (2017) Data-driven documents. [Online]. Available: https://d3js.org/
[14] angularjs.org. (2017) ng. [Online]. Available: https://docs.angularjs.org/api/ng
[15] S.-p. W. Andrew Schwartz, Rick Solis. D3 - a beginner's guide to using d3. [Online]. Available: https://website.education.wisc.edu/~swu28/d3t/index.html
[16] R. Sun. (2014, May) What d3.js is not. [Online]. Available: http://ruoyusun.com/2014/05/26/what-d3js-is-not.html
[17] B. Jacobson. (2016, Jun.) Great interactive data visualization tools in 2016. [Online]. Available: https://constructive.co/insights/6-best-interactive-data-visualization-tools-2016-part-2/
[18] Crypto — node.js v9.2.0 documentation. [Online]. Available: https://nodejs.org/api/crypto.html#crypto_crypto
[19] Passport. Passport. [Online]. Available: http://www.passportjs.org/
[20] CAS. (2015, Jun.) How cas works. [Online]. Available: https://idms.rutgers.edu/cas/how_does_it_work.shtml
[21] Passport. Passport. [Online]. Available: http://www.passportjs.org/docs
[22] puranjay. (2015, Jul.) What css frameworks should you use? [Online]. Available: https://www.csshero.org/css-frameworks-use-comparing-5-popular-css-frameworks/
[23] N. Pettit. (2016, May) Should you use bootstrap or foundation? [Online]. Available: http://blog.teamtreehouse.com/use-bootstrap-or-foundation
[24] W. Blankenship. (2017, Oct.) Bootstrap vs. foundation: Which framework is right for you? [Online]. Available: https://www.upwork.com/hiring/development/bootstrap-vs-foundation-which-framework-is-right-for-you/
[25] L. Bradford. (2015, Jun.) Bootstrap 3 vs. foundation 5: Which front-end framework should you use? [Online]. Available: https://www.codementor.io/codementorteam/bootstrap-3-vs-foundation-5-which-front-end-framework-should-you-use-8s90mhsgn
[26] C. Sevilayha. (2015, Jan.) How to correctly use bootstrapjs and angularjs together. [Online]. Available: https://scotch.io/tutorials/how-to-correctly-use-bootstrapjs-and-angularjs-together
[27] M. Rajput. The pros and cons of choosing angularjs. [Online]. Available: https://jaxenter.com/the-pros-and-cons-of-choosing-angularjs-124850.html
[28] ——. Top reasons why web developers choose angularjs. [Online]. Available: http://www.c-sharpcorner.com/article/top-reasons-why-web-developers-choose-angularjs/
[29] Y. Gidalevitz. Javascript mvc frameworks: Your choice is your future. [Online]. Available: https://www.codal.com/javascript-mvc-frameworks-your-choice-is-your-future/#
[30] T. Volodymyr. Express.js mobile app development: Pros and cons for developers. [Online]. Available: https://jssolutionsdev.com/blog/express-mobile-app-development/
[31] N. Kharchenko. How to choose the best node.js framework: Express.js vs koa.js vs sails.js. [Online]. Available: https://www.upwork.com/hiring/for-clients/choose-the-best-node-js-framework-express-js-vs-koa-js-vs-sails-js/
[32] T. Sharma. Hapi js vs loopback for writing enterprise application. [Online]. Available: https://medium.com/@tkssharma/hapi-js-vs-loopback-for-writing-enterprise-application-b86c86ccfb22
[33] Amazon. Amazon aws. [Online]. Available: https://aws.amazon.com/free
[34] R. Goodwin. Amazon website hosting: Complete aws review (updated 2017). [Online]. Available: https://howtostartablogonline.net/amazon-web-hosting-review/
[35] M. Partners. Drupal pros and cons: An in depth look. [Online]. Available: https://www.mcdpartners.com/news/drupal-pros-and-cons/
[36] O. S. University. Drupal. [Online]. Available: http://is.oregonstate.edu/service/drupal
[37] J. Symonds. My love/hate relationship with heroku. [Online]. Available: http://joshsymonds.com/blog/2012/06/03/my-love-slash-hate-relationship-with-heroku/
[38] R. Shaikh. (2016, Jan.) Salt hash passwords using nodejs crypto. [Online]. Available: https://ciphertrick.com/2016/01/18/salt-hash-passwords-using-nodejs-crypto/
[39] Passport. Passport. [Online]. Available: http://www.passportjs.org/docs/username-password
[40] J. Watkins. Angular - bootstrap ui - datepicker. [Online]. Available: https://codepen.io/joe-watkins/pen/KsAgp