Winter 2018 Progress Report for: Scalable Web Application Framework for Monitoring Energy Usage on Campus

Daniel Schroeder, Aubrey Thenell, Parker Bruni

Abstract

The purpose of this progress report document is to outline the progress made on the Scalable Web Application Framework for Monitoring Energy Usage on Campus project over the past six weeks. Provided in this outline are the accomplishments and problems, our project's goals and purpose, and the current status of our project.

CONTENTS

1 Introduction

1.1 Purpose

Our project is to create a web application to monitor energy use on Oregon State University's campus. The application should serve all the requirements outlined by the client and be easy to use for users of all experience levels. Some specific functionalities that our application should contain are:

- Receive data from Obvius AcquiSuite data acquisition servers and process this data into interpretable graphs.
- Allow administrative users to add buildings and meters to the database as monitoring efforts expand to more buildings on campus.
- Allow users to create unique dashboards and dashboard collections in an effort to organize data into related subsets.
- Have a public facing interface where administrators can produce content for anyone to see.
- Contain modular components with individualized functionality and the ability to share data across components.
- Update graphs being displayed as new data is received.
- Undergo usability testing and produce an interface that is user friendly and easily navigable.
- Embrace AngularJS concepts to inject content to the page as new requests are made.

1.2 Scope

The scope of this report is Winter term 2018 between week 0 and week 6.

1.3 Overview

This document provides a recap of the progress made on our project during the Winter 2018 term. Overall, we have seen some great progress. We went from a basic template website to something that has some great functionality.

2 Contributor: Daniel Schroeder

2.1 Describe where you are currently on the project

Since the end of fall term, I have implemented 3/4 main components of the application with back-end functionality and data-binding working. These components include Buildings, Blocks, and Dashboards. In addition, I generated important mongoose.js and AngularJS code for repeated iterations and element generation of User-stored data and saving documents created by the user. I got Google oAuth working on the application with user objects stored by Google API tokens and create user sessions throughout the application with passport.js.

2.1.1 Buildings

The building components are data models stored in the database with attributes like name, type, id, serial, and array of data points. To generate components that display these buildings, we needed to store the data along with building images and create a page to list them in a logical way. To accomplish this, I scraped the internet for a photo of each building we needed in the application and stored them in our assets folder with a name that corresponds to the building object name that is stored in the database. On the "/buildings" page, a building service retrieves all the building objects from the database which are used as the data model for an AngularJS "ng-repeat" directive to generate identical

components. The outcome is a precise, columned list of card components that link to specific building pages and display the object's name and photo.

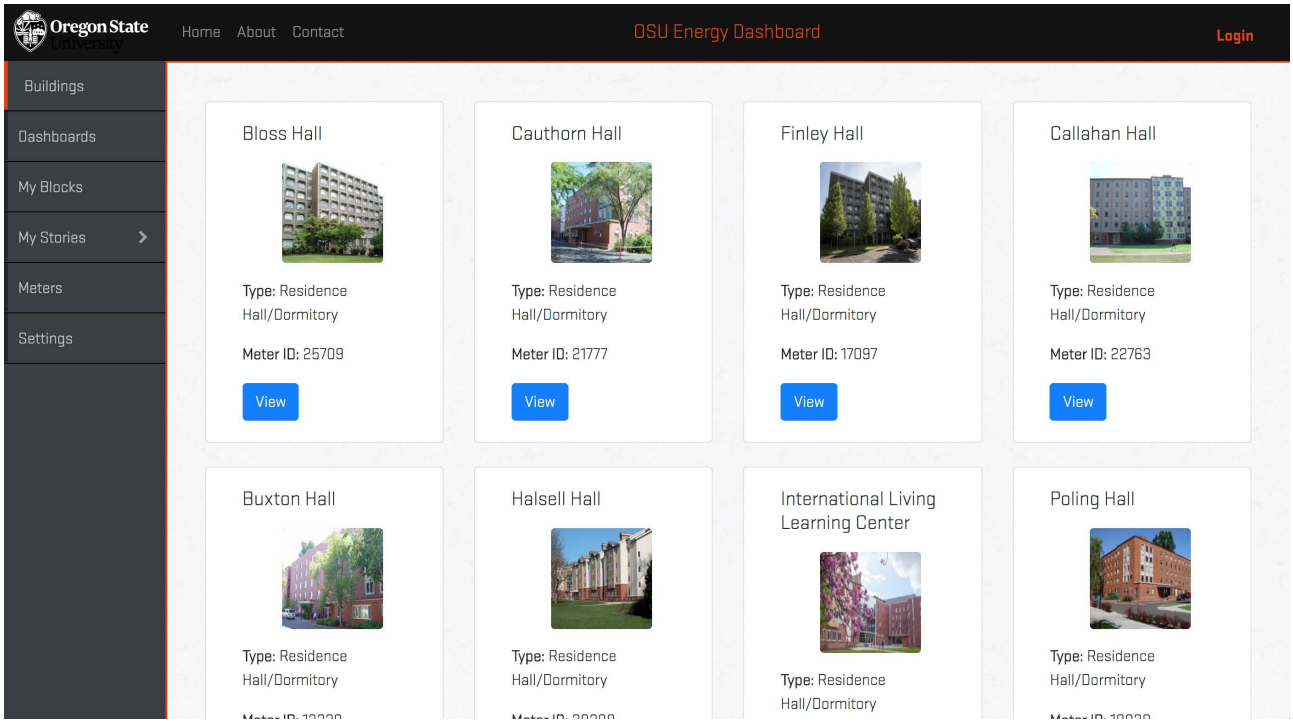

Fig. 1: A screen shot of the building components being listed by the ng-repeat AngularJS directive.

Some clever code I used for the building components includes a regular expression used to parse the building name and convert it to the image address for the photo. As shown below, each building component has an "<img>" tag inside its card which calls a controller function "getImageAddress". This function executes a regular expression in the controller based on the building object that gets passed in from the view.

```
<!--buildings.html-->
<div class="card-body ml-3">
  <h5 class="card-title">{{building.name}}</h5>
  <div class="rounded mx-auto mt-3" style="height: 100px; width: 100px;">
      <img class="rounded" ng-src="{{getImageAddress(building)}}" alt="">
  </div>
  <p class="mt-3"><b>Type: </b>{{building.building_type}}</p>
  <p><b>Meter ID: </b>{{building.meter_id}}</p>
  <a ng-click="viewBuilding(building)" href="#viewBuilding" class="...">View</a>
</div>
```

```
//building-controller.js
$scope.getImageAddress = function(building) {
  return "../assets/buildings/"+building.name.replace(/\s+/g, '-').toLowerCase()+".jpg";
};
```

These building components will be publicly viewable and each building card has a "view" button which links to the individualized building page. These individual pages will contain (they are not yet completed) information about the

building like photos, name, and type, and a personalized report for the specific building with consumption information and charts.

2.1.2 Blocks

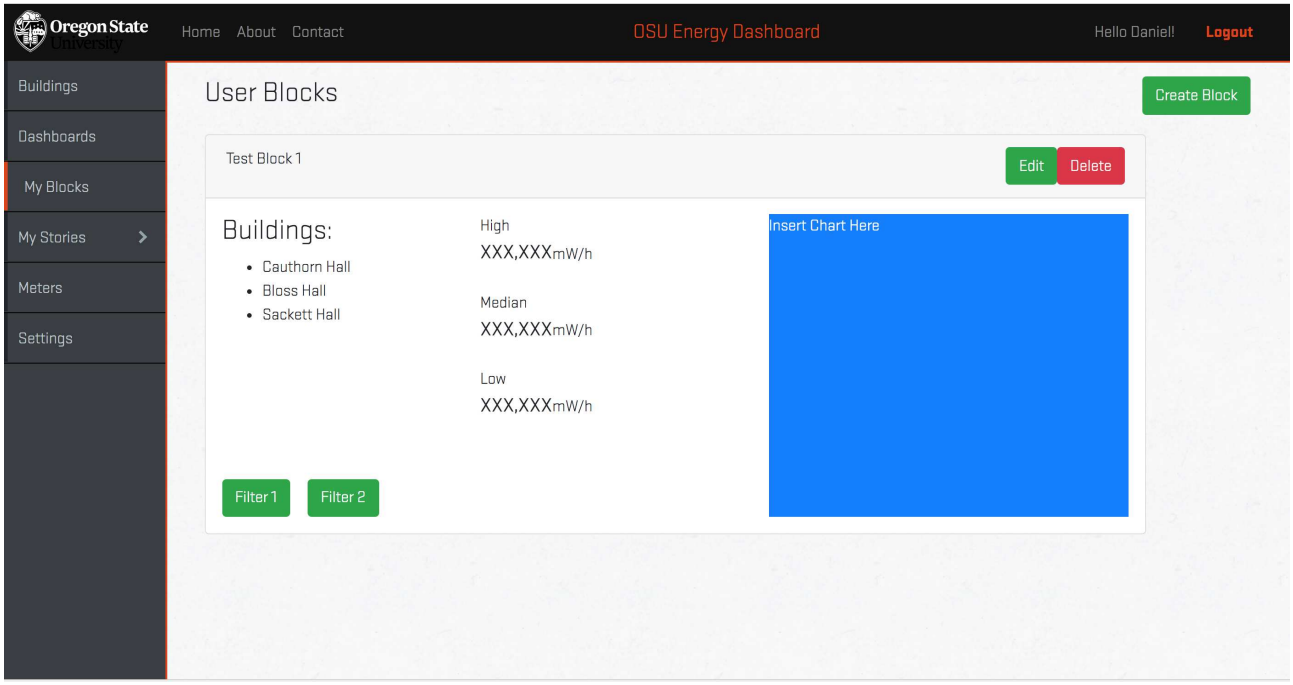

Fig. 2: A screen shot of a block component created by the user with three building objects.

The block components I built are the "building blocks" of user dashboards and the hub for user created graphs/reports. A Block object contains an array of buildings, a graph, and a name. A block generates a specific type of graph based on the building(s) selected and the type of graph the user selected. The D3.js visualizations have not been implemented yet, but each block will retrieve and parse the data for each building it stored in its building array and generate a D3.js visualization. These graphs can range from simple consumption over time line graphs, to comparison graphs between multiple buildings.

To implement these components, I had to create a model schema, build a create form page, and design the UI. Additionally, I needed to create controller functions and AngularJS services to handle data creation, retrieval, and storing. We wanted a way to keep track of which blocks were created by which users, so in addition to a Blocks table in the database, each user has an array of type Block within their user object that gets pushed to and pulled from on creation or deletion. This architecture makes data retrieval much simpler for a few reasons:

- Do not need to iterate through the entire Block table when retrieving user blocks.
- Mongoose.js has a .populate function that populates sub-documents referenced by other documents.
- Block retrieval service can be user-specific.

To elaborate more on this style of NoSQL database management, I will show our User and Block schemas and explain how I implemented the storing and retrieval.

```
var blockSchema = mongoose.Schema({
  name          : String,
  created_by    : {type:mongoose.Schema.ObjectId, ref: 'User'},
  building      : [{type:mongoose.Schema.ObjectId, ref: 'Building'}],
  chart         : String,
  variable      : String
});

var userSchema = mongoose.Schema({
  google            : {
      id            : String,
      token         : String,
      email         : String,
      name          : String
  },
  blocks            : [{type:mongoose.Schema.ObjectId, ref: 'Block'}],
  dashboards        : [{type:mongoose.Schema.ObjectId, ref: 'Dashboard'}]
});
```

The User schema defines a User as having an array of block objects and an array of Dashboard objects (defined by the "[ ]" in the attribute definition). One thing to notice is the mongoose syntax for referencing another schema with the definition mongoose.Schema.ObjectId. This is the mongoose equivalent of setting a foreign key. When we push a document to the User.block array, the Block._id will be stored in this block array, effectively creating what mongoose likes to call a sub-document. More explicitly: "Sub-documents are documents embedded in other documents. In Mongoose, this means you can nest schemas in other schemas"[1]. Creating a "relational database" like this with NoSQL allowed me to generate simple queries in our API route handlers that find a User, populate all the referencing sub-documents, and return the resulting JSON back to the controller. From here, we are able to store references to buildings in the block objects, references to blocks in the dashboard objects, and only populate data when we need it in the application.

2.1.3 Dashboards

The dashboard components are the last thing I worked on and are not yet fully complete. What I wanted to get finished before the midterm milestone was creating dashboards and storing dashboards, which I accomplished. There is a "create-dashboard" form that data-binds all user blocks to a drop down menu using AngularJS directive "ng-options" so that a user is able select which blocks they want to include in the dashboard. In addition, the form has a text-box for a dashboard name and a multi-line text-box for a description attribute, all of which are gathered up by the dashboard controller and passed to an API route-handler that stores the new Dashboard object in the database.

The dashboards followed the same suit as the Block objects as they were stored in an array within the creating User's object as well as in their own table in the database. This produced some difficulties when querying updates and deletes as we had to ensure that both references were updated for a successful return. I still have to implement the "view dashboard" feature which will consist of iterating over each Block stored in the Dashboard and rending its content/graphs in a sequential order. Our dashboards are essentially the same as the "view blocks" page with block components listed one after the other, except it contains a user-defined subset of blocks for a specific analysis.

2.1.4 Important Data-Binding Code

Throughout the application, I have implemented AngularJS directives/services to retrieve and render data to the screen. Here are a few specific code samples I want to share that were either duplicated and used on multiple pages, or I just thought they were impressive implementations.

First is the "ng-repeat" directive we use for buildings, dashboards, and blocks to iterate through a data set and produce multiple HTML elements "for-each" object in the set. This implementation requires a data set to be provided by the controller and an HTML template to be repeated. A particularly interesting example of this was a nested "ng-repeat" used to display all the user blocks, and all the buildings inside each block.

```
1  <!--blocks.html-->
2  <div ng-controller="blockController">
3  <div class="card mb-3" style="width: 90%;" ng-repeat="block in userBlocks">
4    <div class="card-header h-100">
5      <div class="h-100 d-inline-flex">
6        <span class=" align-middle ">
7          {{block.name}}
8        </span>
9      </div>
10     ...
11     <h3>Buildings:</h3>
12     <ul>
13       <li ng-repeat="building in block.building">
14         {{building.name}}
15       </li>
16     </ul>
17   ...
```

On line 13 of the listing, we see that the data model being repeated is being taken from the block object returned by the "ng-repeat" directive on line 3. This essentially allows "double for-loop" style data-binding to occur in the view and which displays all the necessary information to the user with only a couple lines of code.

Next, I wanted to share how I was able to create a functioning relation database with MongoDB and mongoose.js using sub-documents and the mongoose.js ".populate()" function in our API. In order to keep our object sizes small, we only store references (ObjectId's) into model arrays like Block.buildings or User.dashboards. In order to retrieve this data and render it for the user, we need to dereference the ObjectId's in the back-end before returning the object to the controller. To do this, we have to populate the sub-documents so mongoose can retrieve the actual objects from the database and return all the necessary information. I'm going to share the API query for achieving this for the same situation as the nested "ng-repeat" above, as it also required a nested ".populate()" call to dereference sub-documents and sub-sub-documents.

```
1  app.get('/api/getUserBlocks', function(req, res) {
2    User.findOne({_id : req.user._id})
3      .populate({ path: 'blocks',
```

```
4           populate: {path: 'building'}
5       })
6       .exec(function (err, user) {
7         if (err) return handleError(err);
8         res.json(user.blocks); });
9   });
```

As seen on line 4 of the listing, we call a populate from within a populate which dereferences the sub-sub-documents "building" that are being referenced by the sub-document "blocks." This ensures that we have access to the building names and types when returning the user blocks to the controller. If we did not populate these sub-documents, the query would only return the ObjectId's with no relevant information for display.

2.1.5 Authentication

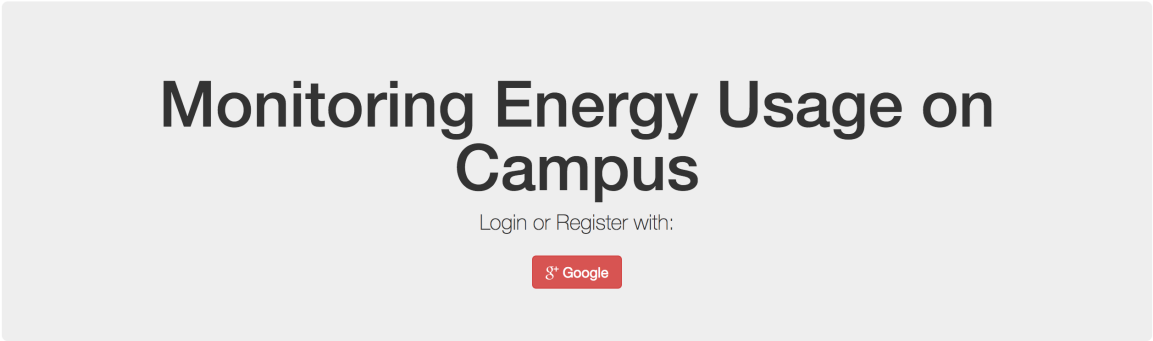

Fig. 3: A screen shot of the login page that uses Google oAuth to authenticate users.

In addition to component functionality and implementation, I was able to get Google oAuth2.0 authentication to work with our application and authenticate users based on Google token. I had our client (Jack Woods) generate a Google API key under an Office of Sustainability Gmail account and used that API key to set up our authentication protocol. I used passport.js as an authentication middleware which handles most of the session authentication and oAuth functionality. I stored the token that Google's API returned in the user object which is what our application uses instead of a password. The next steps to take for authentication is limiting access to certain pages depending on user role and generating "per-page" authentication middleware to track users across pages.

2.1.6 Other Features I Accomplished

Despite the big features mentioned above, other tasks I accomplished since last term are:

- Stored all sensitive information as ENV variables accessible through "dotenv" npm library.
- Created User, Block, Dashboard, and Building services to call API route handlers and access data.
- Formatted "Our Team" section into tri-column sections with colored job-titles and blank avatar images.
- Generated angular routes in the core.js file to create SPA (single page application) protocols for linking and navigation.

- Built the pop-out story menu in sidebar.
- Built side navigation.
- Added all building images in assets folder.
- Created highlighting effects and border removal AJAX functionality for navigation selection.
- Created the dynamic population of “selected buildings” in create block form.
- Created the dynamic population of “selected blocks” in create dashboard form.
- Implemented delete block functionality (button, back-end query, and redirect).
- Implemented delete dashboard functionality (button, back-end query, and redirect).
- Generated mock-up table for data records in individual building pages.
- Created a database configuration file to generate all the buildings with example data.

2.2 Describe what you have left to do

Some things I still need to accomplish:

- Implement story functionality.
- Create “edit” pages for block, dashboard, and story components.
- Filter settings on buildings page.
- Limit viewable content based on user roles and authentication.
- Make public vs. private flag on user items.
- Create public facing application distinction.
- Generate D3.js templates for different graph types.

3 Contributor: Aubrey Thenell

3.1 Describe where you are currently on the project

Since the last progress report, I’ve done a lot of work on the heavier side of the backend. Specifically, I am talking about parsing XML data received by our server as a POST request and then dealing with it appropriately. This mainly revolved the “Building” component, as all the data coming in will be added to that database. I also redesigned the schemas for both the “Building” and “DataEntry” components for how data should be entered into each respective database. I also have helped with the implementation of a couple other components as well, but not nearly to the same extent.

3.1.1 XML Parsing

As mentioned above, I implemented functionality to parse XML data that is sent to our server via POST. When I say parsed, I would like to specify that the XML object is being parsed and converted into a JSON object. As the functionality is currently set up, once the data is parsed, it will call a separate function after this asynchronous parsing function resolves and returns the JSON.

3.1.2 Buildings

One of the first things I needed to develop was a schema for the building and data entries. Changing them from the template to what they are now will serve as a much simpler solution for when adding non hardcoded entries. This section is where I spent most of my hours thus far in the term. Where Dan left off with example building data, I took

over. After the parse, it will check the data to see if there are any duplicates. It will use the AcquiSuite's serial number as well as the timestamp of the data entry. This functionality is especially important because multiple entries for the same data could unevenly weigh some of the averages. There is also the issue that could arise of how to handle the data GET request if more than one entry was returned for any given serial and timestamp coupling. While checking for duplicates, there will be a check to see if the building and meter exists in our database in the first place. If not, it will create one with all the appropriate name and serial. The implementation of the "add building" function is rather flexible. Inside the asynchronous function that checks to see if the building exists yet, if it doesn't exist, will create a special object to be passed as a parameter for the "add building" function. This object will match the same format as the user side "add building" will.

```
Building.findOne({serial: req.body.das.serial}, function (err, doc) {
 if(doc === null){
  var entry = {
   name: req.body.das.devices.device.name,
   building_type: 'Academic',
   serial: req.body.das.serial
  }
```

You may notice in this figure that "building_type" is hard coded. This is because the building is being added to the database through the XML POST data. The only information that is relevant to the nomenclature of building properties is the name and serial / meter_id. The building type isn't included in the XML. I set it to "Academic" as the default because most of the buildings on campus will be academic in nature. If this needs to be changed, the user will be able to edit the entries of each building as needed. Once the object is formed, the "add building" will be called.

```
function addBuildingToDatabase(entry) {
    Building.findOne({name: entry.name}, function (err, docs) {
   if(docs === null){ // ensure building doesn't exist
    var build = new Building();
    build.name = entry.name
    build.building_type = entry.building_type;
    build.serial = entry.serial;
    build.save()
      .catch( err => {res.status(400)
      .send("unable to save to database");})
    console.log("The building '" + entry.name + "' has been added.");
   }
   else
    console.log('Nothing was added');
 });
  };
```

Once a new building is made, or if it already existed, it will take the data from the JSON object and push it into the data entry array of that respective building. It completes this with the following function.

```
data = new DataEntry();
data.meter_id = req.body.das.serial;
 data.timestamp = new Date(pathShortener.time._);
 pathShortener.point.forEach((e,i) => {data.point[i] = e.$;});
 data.save(function(err, savedBlock) {
  if (err)
   throw err;
  else {
   Building.findOneAndUpdate({meter_id: req.body.das.serial},
   {$push:{data_entry: data, timestamp: data.timestamp}},
   {safe: true, upsert: true, new: true},
   (err) =>{if (err) throw(err);})
            }
        });
```

This function is a bit dense at first glance. For the timestamp, we have to cast the "pathShortner.time._" from a string into a date type. This process is generally relatively straightforward. In the POST data, the timestamp itself isn't formatted like you would expect. It is formatted as an object which includes the timezone and the date as "time.$" and "time._" respectfully. Due to the way the XML gets parsed into JSON, the actual time component ends up with a couple of characters appended inside the timestamp. Luckily the "Date()" function in javascript will parse through that and extract only what we need. For the line after that, we simple iterate over all of the data points inside the report and add them to the the data's array. Finally, when we go to save the data entry, we look for the building to be updated and push it directly to the array. The findOneAndUpdate step may not be required if I were to move it within the scope of my first code snippet.

3.1.3 Other

Besides the building component, I also did some minor tweaks to the Block and Dashboard component. Specifically, there was an error with the "addBlock" function and it would crash our console when block components were interacted with.

The issue stemmed from the block being pushed directly to the user's block array, without checking if the user entry was defined. Another issue with it is it called the ".save" function twice for two different databases which caused a "Maximum call stack size exceeded" error. That error occurs when an infinite loop occurs. As a solution, instead of pushing the data directly to the user array, I implemented it to use the "findByIdAndUpdate" function through Mongoose. The below snippet replaces the above section in the "else" scope.

```
User.findByIdAndUpdate(
    { _id: user._id},
    { $push:{blocks: savedBlock}},
    {safe: true, upsert: true, new: true},
    (err) =>{if (err) throw(err);});
```

This function will find an entry in the database and update it with the new block. If the user doesn't exist somehow, it should create a new entry. However, this is extremely unlikely to happen since users must be logged in to even create a block. This method was also implemented into the "Dashboard" component since it had a duplicate problem.

3.2 Describe what you have left to do

- There is still much to do for the backend side of this project. One of the main things I still need to implement is duplicate data prevention in the data entry function. This is currently a work in progress, but unexpected results are currently being returned from the data entry array for the timestamp object.
- I also need to add a frontend way for users to add, edit, and delete buildings. This should be rather simple and similar in functionality to the block, dashboards, and stories. I will be collaborating with Dan on this portion, so we can get a consistent and working function for each.
- Fix building entries. When POST data is sent to our application, it adds the building to the database just fine. However, there is an issue where if there are any other buildings in the database, it will add two objects. One object is a blank shell with the name and meter ID, while the other one contains a placeholder image and the data entry. I think this issue is caused by the image retrieval function in the building-controller file. But it could be due to several reasons.
- Add backend functionality to sort buildings by attributes.
- Implement data sanitation to all of our database entries. As of right now, our database sits completely unprotected minus any built in protections. Although most data entered would require knowledge of the schema to really do much damage. I would still like to play it safe here.

3.3 Describe any problems that have impeded your progress, with any solutions

When I was first trying to get connected to the building database. This was due to my scope being slightly off when I referenced it. Once I figured it out, it all started to work together. Other than that, I haven't run into any major in relation to the project. I did have quite a few staggered hardware problems occur this term. While not directly related to the project, it definitely set my progress back quite a bit.

4 Contributor: Parker Bruni

4.1 Describe where you are currently on the project

At the end of Fall term we had completed our design document and got approval from our client that our design was acceptable and we began work during winter break. So far, I have been assigned to mainly handle the overall look, feel, and front-end design of our project. Progress towards making our application display as a clean and intuitive interface has gone well, without many problems or issues along the way. We ultimately want our application to provide utility, so most of my focus has been for that. I have opted to avoid using flashy or dazzling UI animations or themes to avoid confusion by the user, but the application still needed a clean and modern feel for the user to be comfortable when working. Our color scheme is representative of this, using mostly neutral colors with a few bright colors such as OSU orange as trim to add a little life to the interface.

4.1.1 Fonts and CSS Classes

The font of our application utilizes a directory of Oregon State Marketing approved font files of type ".otf", ".woff", and ".woff2." The font family that these files include are "rufina-stencil-regular" for the main heading of the application and "Stratum2" for other headers. Our body paragraphs will utilize "Open-Sans" for paragraph elements and elements of lesser importance, as per OSU Marketing guidelines. These font files are implemented within our CSS file so that they may be accessed by the HTML elements of our page directly from the CSS file. This is demonstrated in the code below

Listing 1: CSS Implementation language

```
@font-face {
  font-family: 'rufina-stencil-regular';
  src: url('assets/fonts/rufina-stencil-regular.otf') format('opentype'),
       url('assets/fonts/rufina-stencil-regular.woff') format('opentype'),
       url('assets/fonts/rufina-stencil-regular.woff2') format('opentype');
}

.rufina-stencil-regular {
 font-family: 'rufina-stencil-regular';
}
```

Listing 2: HTML Implementation language

```
<h1 class="rufina-stencil-regular m-0">Oregon State University</h1>
```

As you can see from the code, the HTML element is assigned a class which has the desired font associated with it. Each font that we have available can be implemented in an element by adding the class name, essentially functioning as an "argument" by which that element should behave. In this way, we can simply add arguments to our elements via class names to give it certain CSS attributes without having to hard code the CSS style directly into the element. This method is used in our application for many different types desired CSS attributes, such as font, alignment, background-color, position, etc. It allows us great control when customizing CSS attributes without needing to add a specific class for every element. Many of our HTML elements use this technique but utilizing Bootstrap libraries as well, such as generating CSS for our login button element.

4.1.2 Top Navigation

Our application top navigation bar has various elements within it. Specifically, in order from left to right, the top navigation bar contains a application logo, selectable elements that redirect the user to our "Home", "About", and "Contacts" page, our applications title, and our login button. The top navigation bar is a fixed size and position on the page and will exist on all pages until the application is put into presentation mode. It's background color is a very dark gray to achieve a modern look and contrast with the white body of our page and the lighter gray side navigation bar.

4.1.3 Logo

Fig. 4: Our current logo for the application, located in the left-most region of the top nav.

In the current version of our application, our logo has been decided by our client to be an Oregon State official logo. The logo was requested and has been provided for us in the format of a ".svg" file by our client. Having the logo as a scalable vector graphics file will allow the logo adjust to the size of its parent element without loss of resolution or quality if the parent element is to scale up. We do not necessarily have to worry about this as the parent element of our logo will be a fixed size because our top navigation is a fixed size. I have requested to our client that we get a logo that is tailored to our website by a member of the OSU marketing team because it would be much more fitting that our logo includes our application name. Another reason to get a customized logo is because the font of the logo contains black letters which blend in with the dark background of our top navigation bar.

4.1.4 Selectable Elements

The top navigation bar also includes selectable elements so the user can navigate the pages. The elements text highlights on hover but perform no special UI functions when clicked.

4.1.5 Home

The first element is the "Home" tab. When this tab is clicked, the user is redirected to the homepage of our application. This page contains the title of our application as well as a Google Maps API angular.js implementation. The map will contain a KML layer of all of the buildings with metering systems installed, and when a user selects one of these buildings from the map, they will be redirected to that buildings energy dashboard page. Ideally, this homepage will be used mostly by the public where our application will be displayed on a monitor located in buildings on campus.

4.1.6 About

Our about page contains information about the goal of the application, the history of the Office of Sustainability, the mission of of the Office of Sustainability, and links to the social media websites related to the Office of Sustainability.

4.1.7 Contact

Our about page contains the basic contact information of a select few OSU Office of Sustainability members as well as the developers of the application.

4.1.8 Side Navigation

The side navigation was implemented by another member of our team, but I have contributed to the overall look and feel of the side navigation. I have added small animations that indicate to the user which tab has been selected. The first animation fluidly indents the title of the tab when it is being displayed by the body of our page. This is achieved by increasing the "padding-left" of the element from 0px to 6px over the course of 0.2 seconds. The second animation changes the left side border from dark gray to OSU orange over the course of 0.2 seconds. The code and images below demonstrate this effect.

Fig. 5: Side navigation tab selection CSS (orange border and text indent).

Listing 3: CSS Implementation language

```
.custom-nav-item div{
    border-left: 4px solid;
```

```
3
4  }
5  .active-nav-item div{
6      border-left: 4px solid #DC4405;
7      -webkit-transition: all 0.3s ease-in-out 0s;
8      transition: all 0.3s ease-in-out 0s;
9  }
10 .active-nav-item .custom-nav-link{
11     padding-left: 6px;
12  -webkit-transition: all 0.2s ease-in-out 0s;
13     transition: all 0.2s ease-in-out 0s;
14 }
```

These combined CSS animations give modern feel yet clear indication of which tab is selected.

4.2 Describe what you have left to do

Some things that I still need to accomplish:

- Add KML layer to Google Maps API
- Change body font
- Adjust top nav title design
- Obtain customized logo
- Format Buildings page CSS and UI
- Format Dashboards page CSS and UI
- Format My Stories page CSS and UI
- Format Meters page CSS and UI
- Format Settings page CSS and UI
- Add “Presentation Mode” to remove navigation bars and create a more presentable view of application for public access
- Create an official color scheme with set hex color values

4.3 Describe any problems that have impeded your progress, with any solutions

One problem that I had was with our Google Maps API. I had followed a tutorial for adding the Google Maps API but the map was not functioning. I thought this could be because I did not have a valid Map “key” but I tried many different valid keys and this was not the case. I transferred the code that I was using from our application to a simple test implementation and the code worked. For whatever reason, the format of our application was breaking the map. This could be due to our page being a MEAN stack single page application. My solution was to simply look for tutorials on how to implement a Google Map on this style of application. I found a good example and adjusted the code to fit our website. Essentially, I needed to make a controller for this API and eventually the map was functional and displaying properly.

5 Current State of the Project:

We have still been doing local development for the most part. We do have an Amazon EC2 instance spun up at this point, but we would like to not burden our instance with an incomplete software. We will be pushing known stable versions of our application from our master branch to our production branch. We want to make sure our application is working before that. Something as simple as a memory leak occurring on our instance could begin racking up a data bill for our client quickly.

REFERENCES

[1] Mongoose.js, “Sub docs.”