Spring 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 2
1.1 Purpose . . . 2
1.2 Overview . . . 2

2 Contributor: Daniel Schroeder 2
2.1 Describe where you are currently on the project . . . 2
2.1.1 Buildings . . . 2
2.1.2 Blocks . . . 4
2.1.3 Dashboards . . . 6
2.1.4 Important Data-Binding Code . . . 6
2.1.5 Data Retrieval . . . 7

3 Contributor: Aubrey Thenell 12

4 Contributor: Parker Bruni 12

5 Current State of the Project: 12

6 Problems that have impeded our progress 12
6.1 Describe what you have left to do . . . 12

References 13

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 Overview

This document provides a recap of the progress made on our project during the Spring 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

The application is complete except for one main data retrieval function which "subtracts" two meters in order to get the total consumption for a specific building. We are still debating the semantics about how to implement this but will have it done bu expo without a doubt.

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.

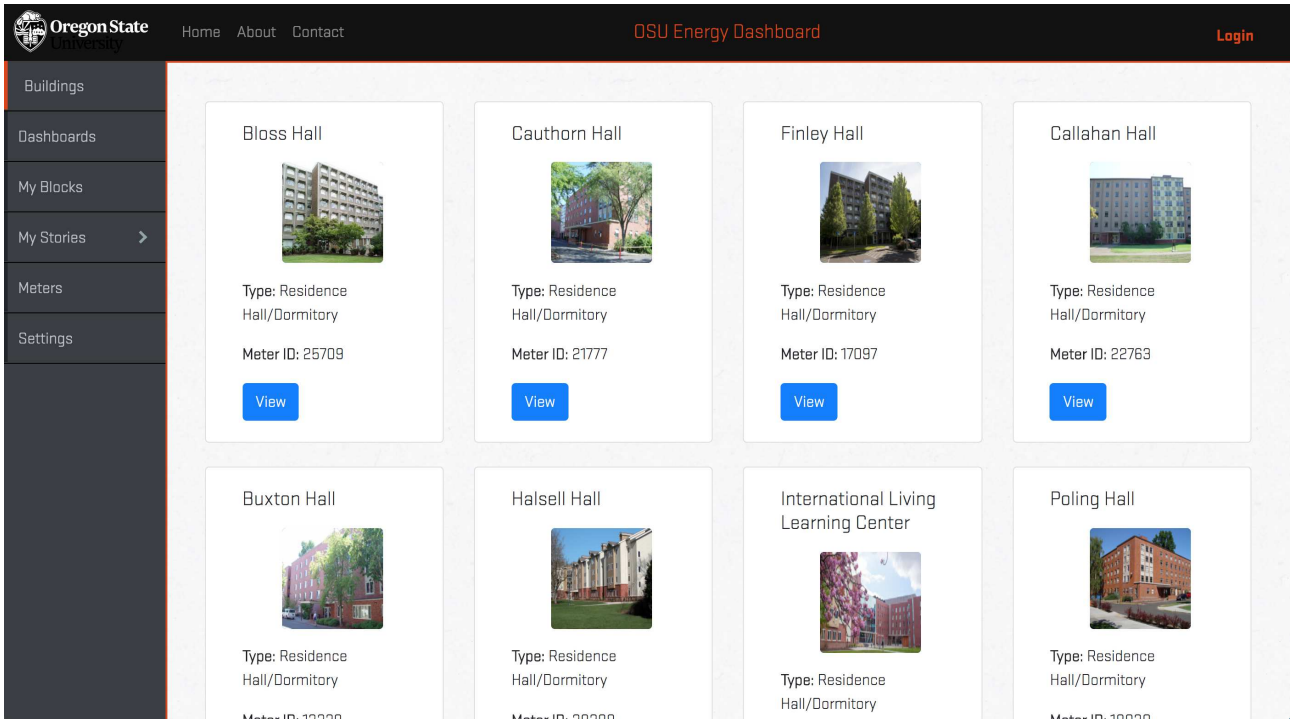

Fig. 1: A screen shot of the building components being listed by the ng-repeat AngularJS directive.

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 contain information about the building like photos, name, and type, and a personalized report for the specific building with consumption information and charts. Below is an example of a single building's page.

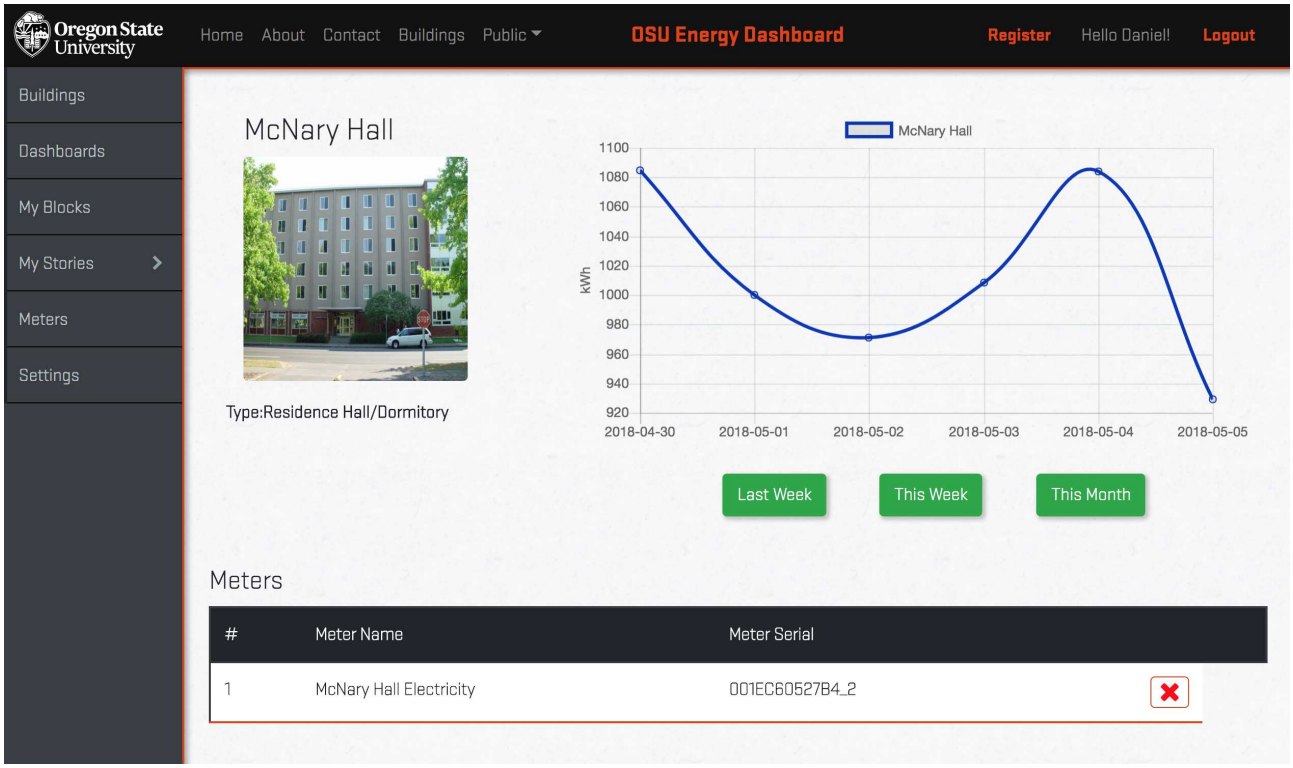

Fig. 2: A screen shot of the building page for McNary Hall.

2.1.2 *Blocks*

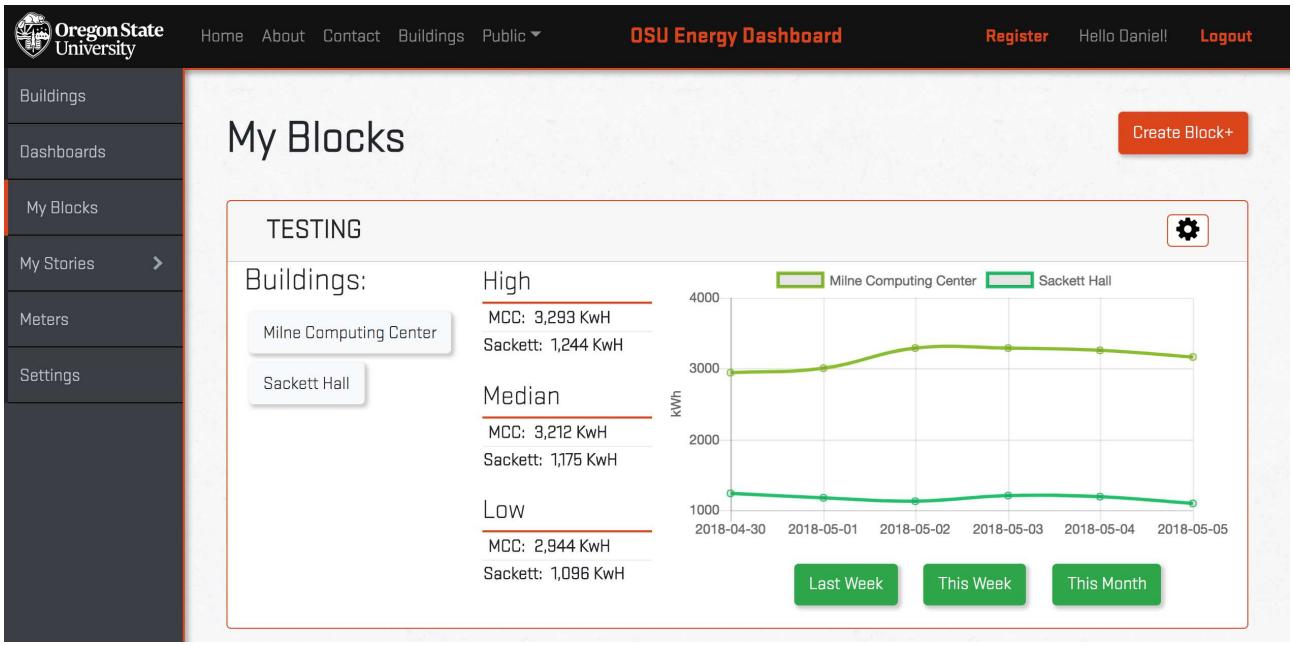

Fig. 3: A screen shot of a block component created by the user with three building objects.

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.

```
<!--blocks.html-->
<div ng-controller="blockController">
<div class="card mb-3" style="width: 90%;" ng-repeat="block in userBlocks">
  <div class="card-header h-100">
    <div class="h-100 d-inline-flex">
      <span class=" align-middle ">
        {{block.name}}
      </span>
    </div>
    ...
    <h3>Buildings:</h3>
    <ul>
      <li ng-repeat="building in block.building">
        {{building.name}}
      </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 Data Retrieval

I wanted to add a code listing of our new data retrieval algorithm as it has changed significantly to include date formatting, data validation for correct value ranges, and summations of multiple meters.

```
1   app.get('/api/getBuildingData', function (req, res) {
2     var match;
3     if (req.query && req.query.start && req.query.end) {
4       match = {
5         timestamp: {
6             $lt: req.query.end,
7             $gte: req.query.start
8         }
9       }
```

```
10      } else {
11        match = {};
12      }
13
14      Building.find({
15        _id: {
16          $in: req.query.buildings
17        }
18        })
19        .populate({
20          path: 'data_entries',
21          match: match, //THIS WORKS TO FILTER DATES
22          select: 'id'
23        })
24        .exec(function (err, dataEntries) {
25          var buildings = [];
26          dataEntries.forEach(function(b){
27            buildings.push({name: b.name, building_id: b._id});
28          });
29          if (err) {
30            res.jsonp({
31              building: null
32            });
33          } else {
34            DataEntry.find({
35              _id: {
36                $in: [].concat.apply([], dataEntries.map(d => d.data_entries))
37              },
38              meter_id: {
39                $in: [].concat.apply([], dataEntries.map(d => d.meters))
40              }
41            })
42            .select({
43              point: {
44                $elemMatch: {
45                  name: "Accumulated Real Energy Net"
46                },
47              }
48            })
49            .sort('timestamp')
50            .select('meter_id timestamp point.value building')
51            .exec(function (err, datapoints) {
52            if (err) {
53                console.log(err);
54            } else {
```

```
55          var meters = [].concat.apply([], dataEntries.map(d => d.meters));
56
57          var temp = [];
58          meters.forEach(function (meter) {
59            var start = new Date(req.query.start);
60            var end = new Date(req.query.end);
61            var array = datapoints.filter(entry => entry.meter_id.toString() === meter.
                  toString());
62            while (start.toISOString().substring(0, 10) < end.toISOString().substring(0,
                  10)) {
63              var daily = array.filter(x => {
64                if (x)
65                  return x.timestamp.substring(0, 10) == start.toISOString().substring(0,
                        10);
66              });
67              if(daily.length>0){
68                var end_index = 1;
69                var start_index = 0;
70                var val = Math.abs(daily[daily.length - end_index].point[0].value) - Math
                      .abs(daily[start_index].point[0].value);
71                // start by decreasing the end value
72                var startflag = 0;
73                while(val < 0 || val > 10000){
74                  if(startflag == 0){
75                    end_index += 1;
76                    if(daily[daily.length - end_index].point[0] && daily[start_index].
                          point[0]){
77                      val = Math.abs(daily[daily.length - end_index].point[0].value) -
                            Math.abs(daily[start_index].point[0].value);
78                    }
79                    startflag = 1;
80                  }
81                  else{
82                    start_index += 1;
83                    if(daily[daily.length - end_index].point[0] && daily[start_index].
                          point[0]){
84                      val = Math.abs(daily[daily.length - end_index].point[0].value) -
                            Math.abs(daily[start_index].point[0].value);
85                    }
86                    startflag = 0;
87                  }
88                }
89                temp.push({
90                    building_id : daily[0].building,
91                    meter_id: meter,
```

```
                    date: daily[0].timestamp.substring(0, 10),
                    val: val
                });
            }
            start.setDate(start.getDate() + 1);
        }
    });
    //A check for Milne to Sum Values
    if(buildings.filter(n => n.name === "Milne Computing Center").length > 0){
        let milne_id = buildings.filter(n => n.name === "Milne Computing Center")[0].
            building_id;
        // add values with common timestamps
        let milne = temp.filter(d => d.building_id.toString() === milne_id.toString());
        temp = temp.filter(d => d.building_id.toString() !== milne_id.toString());

        let vals = milne.reduce((prev, curr) => {
            let count = prev.get(curr.date) || 0;
            prev.set(curr.date, curr.val + count);
            return prev;
        }, new Map());
        [...vals].map(([key, value]) => {
            return {key, value}
        }).forEach(function(point){
            temp.push({building_id: milne_id, date: point.key, val: point.value})
        })
    }
    else if(buildings.filter(n => n.name === "Memorial Union").length > 0){
        let union_id = buildings.filter(n => n.name === "Memorial Union")[0].
            building_id;
        // add values with common timestamps
        let union = temp.filter(d => d.building_id.toString() === union_id.toString());
        temp = temp.filter(d => d.building_id.toString() !== union_id.toString());

        let vals = union.reduce((prev, curr) => {
            let count = prev.get(curr.date) || 0;
            prev.set(curr.date, curr.val + count);
            return prev;
        }, new Map());
        [...vals].map(([key, value]) => {
            return {key, value}
        }).forEach(function(point){
            temp.push({building_id: union_id, date: point.key, val: point.value})
        })
    }
    else if(buildings.filter(n => n.name === "Nash Hall").length > 0){
```

```
135             let nash_id = buildings.filter(n => n.name === "Nash Hall")[0].building_id;
136             // add values with common timestamps
137             let nash = temp.filter(d => d.building_id.toString() === nash_id.toString());
138             temp = temp.filter(d => d.building_id.toString() !== nash_id.toString());
139
140             let vals = nash.reduce((prev, curr) => {
141                 let count = prev.get(curr.date) || 0;
142                 prev.set(curr.date, curr.val + count);
143                 return prev;
144             }, new Map());
145             [...vals].map(([key, value]) => {
146                 return {key, value}
147             }).forEach(function(point){
148                 temp.push({building_id: nash_id, date: point.key, val: point.value})
149             })
150         }
151         else if(buildings.filter(n => n.name === "Kelley Engineering Center").length > 0){
152             let kelley_id = buildings.filter(n => n.name === "Kelley Engineering Center")
                    [0].building_id;
153             // add values with common timestamps
154             let kelley = temp.filter(d => d.building_id.toString() === kelley_id.toString()
                    );
155             temp = temp.filter(d => d.building_id.toString() !== kelley_id.toString());
156
157             let vals = kelley.reduce((prev, curr) => {
158                 let count = prev.get(curr.date) || 0;
159                 prev.set(curr.date, curr.val + count);
160                 return prev;
161             }, new Map());
162             [...vals].map(([key, value]) => {
163                 return {key, value}
164             }).forEach(function(point){
165                 temp.push({building_id: kelley_id, date: point.key, val: point.value})
166             })
167         }
168         else if(buildings.filter(n => n.name === "McNary Hall").length > 0){
169           let mcnary_id = buildings.filter(n => n.name === "McNary Hall")[0].building_id;
170           // add values with common timestamps
171           let mcnary = temp.filter(d => d.building_id.toString() === mcnary_id.toString());
172           temp = temp.filter(d => d.building_id.toString() !== mcnary_id.toString());
173
174           let vals = mcnary.reduce((prev, curr) => {
175               let count = prev.get(curr.date) || 0;
176               prev.set(curr.date, curr.val + count);
177               return prev;
```

```
178         }, new Map());
179         [...vals].map(([key, value]) => {
180             return {key, value}
181         }).forEach(function(point){
182             temp.push({building_id: mcnary_id, date: point.key, val: point.value})
183         })
184       }
185       res.jsonp(temp);
186   }
187 });
188 }
189 });
190 });
```

Although it is not very interesting as a large blob of code, the effect it has on our application with data filtering, formatting, and retrieval is substantially important.

3 Contributor: Aubrey Thenell

4 Contributor: Parker Bruni

5 Current State of the Project:

6 Problems that have impeded our progress

When our group reconvened after Winter Break, we had our application's minimum viable product completed with all requirements satisfied as was due. We then got new information about the XML electricity data that was being sent from the data acquisition servers and how the data was being handled. New information detailed that multiple energy meters could be wired to the AcquiSuite and be sent through successive posts with differing "address" fields. This caused complications with our back-end as we needed to change the way we store data points relative to buildings. New information suggested that some buildings require the sum of two separate energy meter readings (i.e. North and South Meters) and some buildings required the difference of two meter readings (i.e. Complex minus Dining Hall yields residence hall consumption).

6.1 Describe what you have left to do

Some things I still need to accomplish:

- Complete meter subtraction for monitoring McNary residence hall consumption.

[1]

REFERENCES

[1] Mongoose.js, “Sub docs.”