Spring 2018 Progress 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 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 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.

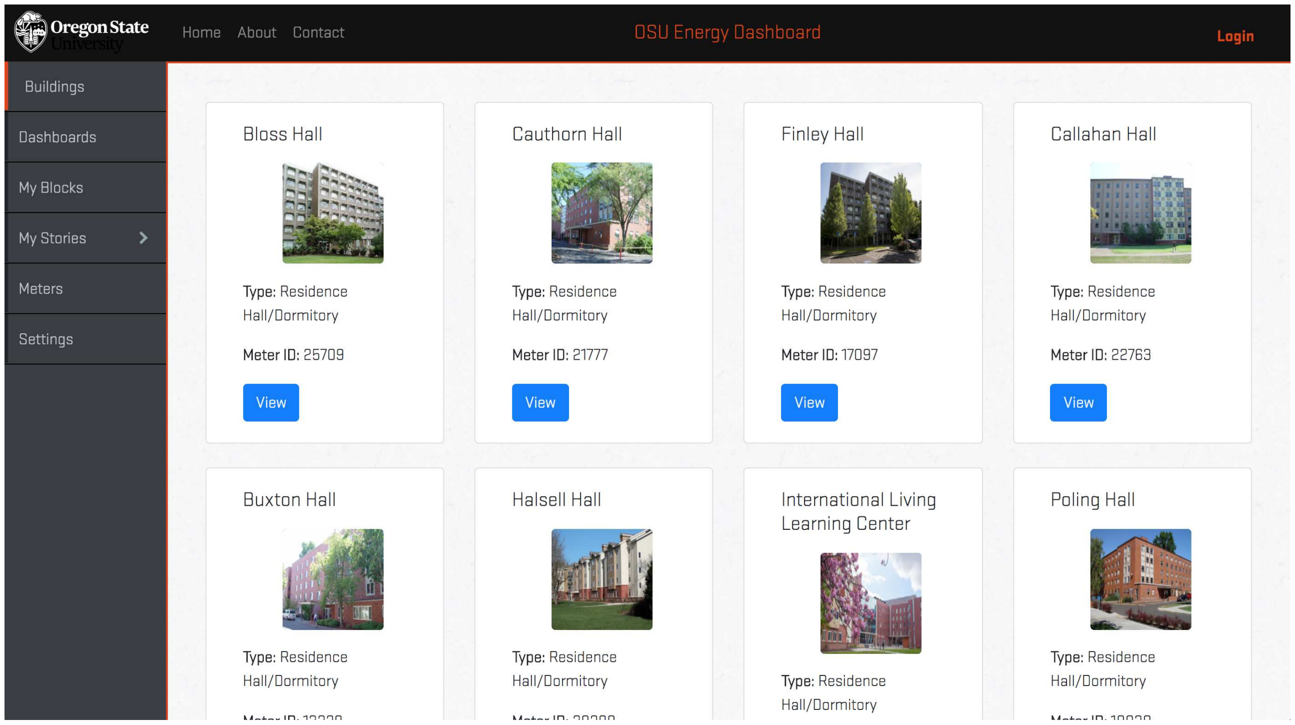
1.3 Describe where you are currently on the project

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. The UI an UX designs have been essentially fully implemented into the application and the various UI/UX elements have been tested for to confirm proper functionality. The color scheme has been solidified and all elements have been adjusted to fit this scheme appropriately. Animations have been added and tested and give the website the desired aesthetic appeal and modern feel to make the usage of the application more pleasing and presentable. The top navigation pages have been essentially finalized, and the professional head shots of all members involved have been added.

2 CONTRIBUTOR: DANIEL SCHROEDER

2.0.1 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.



Bloss Hall
Type: Residence Hall/Dormitory
Meter ID: 25709

Cauthorn Hall
Type: Residence Hall/Dormitory
Meter ID: 21777

Finley Hall
Type: Residence Hall/Dormitory
Meter ID: 17097

Callahan Hall
Type: Residence Hall/Dormitory
Meter ID: 22763

Buxton Hall
Type: Residence Hall/Dormitory

Halsell Hall
Type: Residence Hall/Dormitory

International Living Learning Center
Type: Residence Hall/Dormitory

Poling Hall
Type: Residence Hall/Dormitory

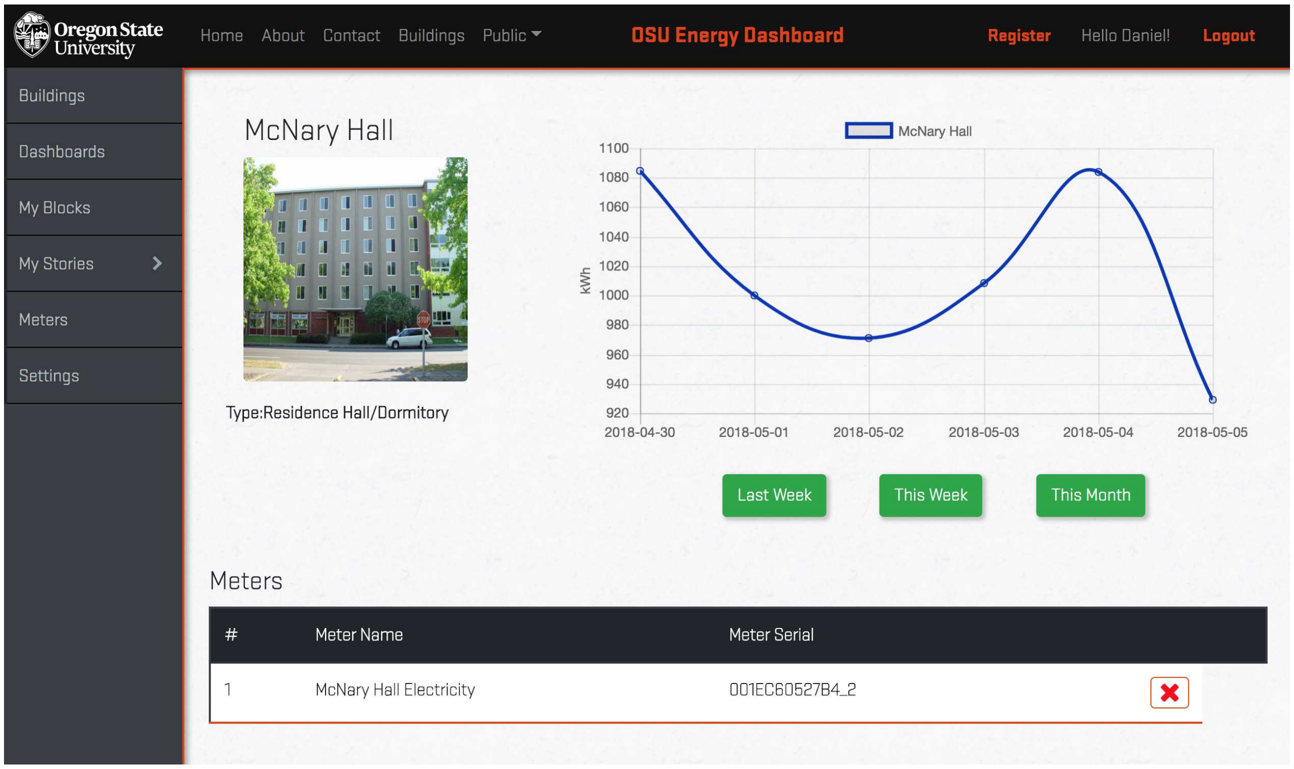
Fig. 1: A screen shot of the building components being listed by the ng-repeat AngularJS directive.

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.

```html
<!--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>
```

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

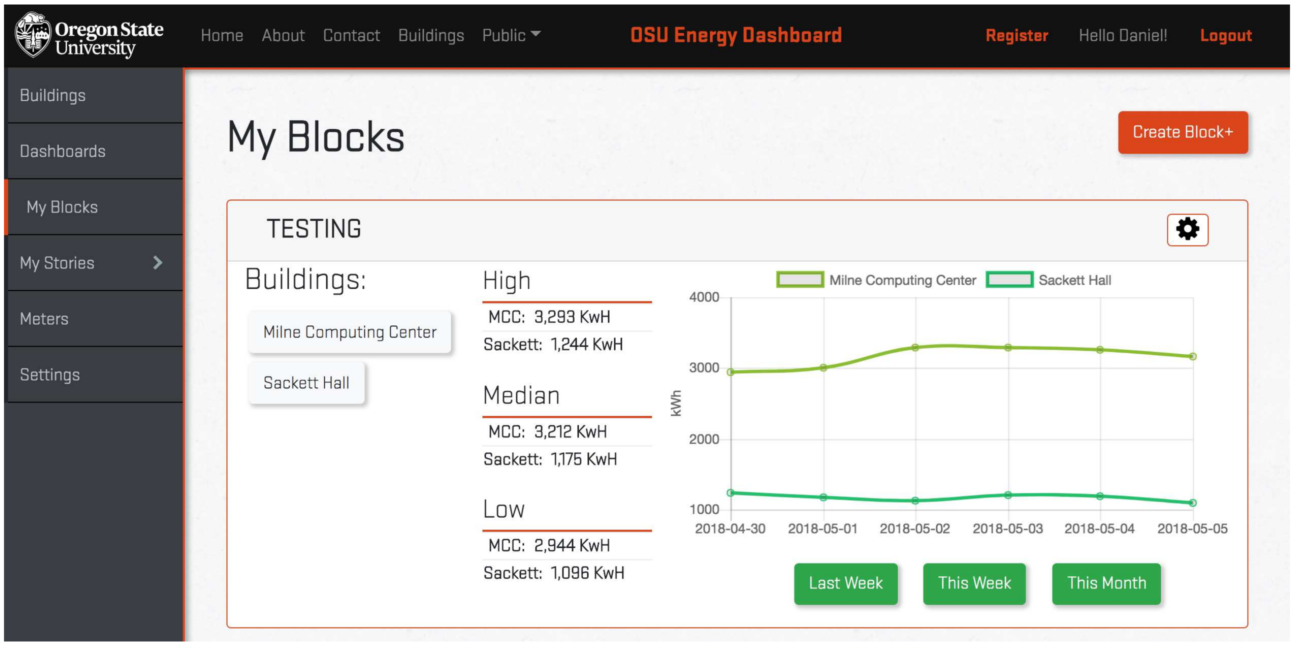
These building components will be publicly viewable and each building card has a "view" button which links to the individualized building page. These individual pages contain information about the building like photos, name, and type, and a personalized report for the specific building with consumption information and charts. Below is an example of a single building's page.



#	Meter Name	Meter Serial
1	McNary Hall Electricity	001EC60527B4_2

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

2.0.2 Blocks



Buildings:

Milne Computing Center

Sackett Hall

High

MCC: 3,293 KwH

Sackett: 1,244 KwH

Median

MCC: 3,212 KwH

Sackett: 1,175 KwH

Low

MCC: 2,944 KwH

Sackett: 1,096 KwH

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.

2.0.3 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.

2.0.4 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.

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

2.0.5 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.

```
 app.get('/api/getBuildingData', function (req, res) {
   var match;
   if (req.query && req.query.start && req.query.end) {
     match = {
       timestamp: {
           $lt: req.query.end,
           $gte: req.query.start
       }
     }
   } else {
     match = {};
   }

   Building.find({
     _id: {
       $in: req.query.buildings
     }
     })
     .populate({
       path: 'data_entries',
       match: match, //THIS WORKS TO FILTER DATES
       select: 'id'
     })
     .exec(function (err, dataEntries) {
       var buildings = [];
       dataEntries.forEach(function(b){
         buildings.push({name: b.name, building_id: b._id});
       });
       if (err) {
         res.jsonp({
           building: null
         });
       } else {
          DataEntry.find({
            _id: {
              $in: [].concat.apply([], dataEntries.map(d => d.data_entries))
            },
            meter_id: {
              $in: [].concat.apply([], dataEntries.map(d => d.meters))
            }
          })
          .select({
            point: {
              $elemMatch: {
                name: "Accumulated Real Energy Net"
              },
            }
          })
          .sort('timestamp')
          .select('meter_id timestamp point.value building')
          .exec(function (err, datapoints) {
          if (err) {
              console.log(err);
          } else {
          var meters = [].concat.apply([], dataEntries.map(d => d.meters));

          var temp = [];
          meters.forEach(function (meter) {
            var start = new Date(req.query.start);
            var end = new Date(req.query.end);
            var array = datapoints.filter(entry => entry.meter_id.toString() === meter.
                toString());
            while (start.toISOString().substring(0, 10) < end.toISOString().substring(0,
                10)) {
              var daily = array.filter(x => {
                if (x)
                  return x.timestamp.substring(0, 10) == start.toISOString().substring(0,
                      10);
              });
              if(daily.length>0){
                var end_index = 1;
                var start_index = 0;
                var val = Math.abs(daily[daily.length - end_index].point[0].value) - Math
                    .abs(daily[start_index].point[0].value);
                // start by decreasing the end value
                var startflag = 0;
                while(val < 0 || val > 10000){
                  if(startflag == 0){
```

```
                  end_index += 1;
                  if(daily[daily.length - end_index].point[0] && daily[start_index].point[0]){
                      val = Math.abs(daily[daily.length - end_index].point[0].value) - Math.abs(daily[start_index].point[0].value);
                  }
                  startflag = 1;
                }
                else{
                  start_index += 1;
                  if(daily[daily.length - end_index].point[0] && daily[start_index].point[0]){
                      val = Math.abs(daily[daily.length - end_index].point[0].value) - Math.abs(daily[start_index].point[0].value);
                  }
                  startflag = 0;
                }
              }
              temp.push({
                  building_id : daily[0].building,
                  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){
        let nash_id = buildings.filter(n => n.name === "Nash Hall")[0].building_id;
        // add values with common timestamps
        let nash = temp.filter(d => d.building_id.toString() === nash_id.toString());
        temp = temp.filter(d => d.building_id.toString() !== nash_id.toString());

        let vals = nash.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: nash_id, date: point.key, val: point.value})
        })
    }
    else if(buildings.filter(n => n.name === "Kelley Engineering Center").length > 0){
        let kelley_id = buildings.filter(n => n.name === "Kelley Engineering Center")
            [0].building_id;
        // add values with common timestamps
        let kelley = temp.filter(d => d.building_id.toString() === kelley_id.toString()
            );
        temp = temp.filter(d => d.building_id.toString() !== kelley_id.toString());

        let vals = kelley.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: kelley_id, date: point.key, val: point.value})
        })
    }
    else if(buildings.filter(n => n.name === "McNary Hall").length > 0){
      let mcnary_id = buildings.filter(n => n.name === "McNary Hall")[0].building_id;
      // add values with common timestamps
      let mcnary = temp.filter(d => d.building_id.toString() === mcnary_id.toString());
      temp = temp.filter(d => d.building_id.toString() !== mcnary_id.toString());

      let vals = mcnary.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: mcnary_id, date: point.key, val: point.value})
      })
    }
    res.jsonp(temp);
}
});
}
});
});
```

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

3.1 XML Parsing

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.

3.2 Buildings and Meters

Buildings and meters are fully functioning and can be added to the database via the '/api/addBuilding' and '/api/addMeter' function calls respectively. Shown below is a snippet demonstrating adding a building. Functionally both the building and meter functions look and work about the same way. Buildings can only be added to the database through this function on the client server, compared to meters when can be added by both servers.

```
app.post('/api/addBuilding', function (req, res) {
       var building = new Building();
       building.name = req.body.name;
       building.building_type = req.body.building_type;
       building.meters = req.body.meters;
       building.save(function (err, savedBuilding) {
           if (err)
               throw err;
           else {
               savedBuilding.meters.forEach(meter => {
                   updateOldBuildingMeters(meter, savedBuilding)
                       .then(addMeter(meter, savedBuilding))
               });
               res.json(savedBuilding);

           }
       });
   });
```

Once a building or meter is added to the database, it can be updated like any of the other components. Shown below is the function to update buildings. In the else block towards the bottom of the function, you will notice that it makes two function calls. What those functions do is remove any meters from the old building and add them to the new building. By doing this, it ensures that data entries will only ever go to one building at a time and this prevents duplicate and incorrect data from entering our database.

```
app.post('/api/updateBuilding', function (req, res) {
       Building.findByIdAndUpdate({
           _id: req.body._id
       }, {
           $set: {
               'name': req.body.name,
               'building_type': req.body.building_type,
               'meters': req.body.meters
               // instead of pushing meters here, might do similar function call like in
                    addBuilding
           }
       }, {
           safe: true,
           upsert: true,
            new: true
        }, function (err, meter) {
            if (err)
                throw (err);
            else {
                req.body.meters.forEach(meter => {
                    updateOldBuildingMeters(meter._id, req.body)
                        .then(addMeter(meter._id, req.body))
                });

            }
        });
        res.json(req.body);
    });
```

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.

```
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);
                }
            });
        });
```

This function will iterate over the received data entries from a given AcquiSuite meter and add each one to the DataEntry database as well as push a reference to that DataEntry entry to the connected building's DataEntry array.

3.3 Email

There are a few email related functions that are fully working within sandbox mode. This means that we can only send to AWS verified email addresses. Once the client decides he wants to move to production, then email functionality will work on any and all email addresses.

3.3.1 Registration

The registration allows the Administrative users of the site to invite other users with various permission levels. Below is the full function that shows the AWS format for sending emails

```
app.post('/api/emailRegistration', function (req, res) {
        AWS.config.update({
            region: 'us-west-2'
        });
        var credentials = new AWS.EnvironmentCredentials('AWS');
        credentials.accessKeyId = process.env.AWS_ACCESS_KEY_ID
        credentials.secretAccessKey = process.env.SECRET_ACCESS_KEY
        AWS.config.credentials = credentials;
        console.log(req.body);
        console.log('Something');
        User.findOneAndUpdate({
            'google.email': req.body.email
        }, {
            $set: {
                accountAccess: req.body.access
            }
        }, {
            upsert: true,
            new: true
        }, (err, doc) => {
            console.log('callback')
            if (err) throw err;
            if (doc) {
                console.log(doc);
                if (doc.accountAccess !== req.body.access) {
                    console.log(doc)
                }

            }
        });
        var params = {
            Destination: { /* required */
                CcAddresses: [],
                ToAddresses: [req.body.email]
            },

            Message: { /* required */
                Body: { /* required */
                    Html: {
                        Charset: "UTF-8",
                        Data: "<h1>This is an E-mail from the application</h1><br> <h4>
                            Please click the link below to be taken there.</h4><br>" +
                            "<a href=\"http://localhost:3000/login/\">Click me!</a>"
                    },
                    Text: {
                        Charset: "UTF-8",
                        Data: "TEXT_FORMAT_BODY"
                    }
                },
                Subject: {
                    Charset: 'UTF-8',
                    Data: 'Test email from AWS'
                }
            },
            Source: process.env.TEST_EMAIL_USER,
            /* required */
            ReplyToAddresses: [],
        };
        var sendPromise = new AWS.SES({
            apiVersion: '2010-12-01'
        }).sendEmail(params).promise();

        // Handle promise's fulfilled/rejected states
        sendPromise.then(
            function (data) {
                console.log(data.MessageId);
            }).catch(
            function (err) {
                console.error(err, err.stack);
            });
        res.json({
            message: "success"
        });
    });
```

I felt it was important to list the whole function because the other two email based functions will use this same format, but with different data.

3.3.2 Usage Monitoring

One of the core requirements was to send out email alerts whenever a meter's consumption was way higher than normal. This function does exactly that. To do this, it takes the average energy usage of the past week and sums one standard deviation. Then it takes the average consumption for the past day. If the of the past day is one standard deviation above the weekly consumption, an email will be sent out. This function is automatically called once per day.

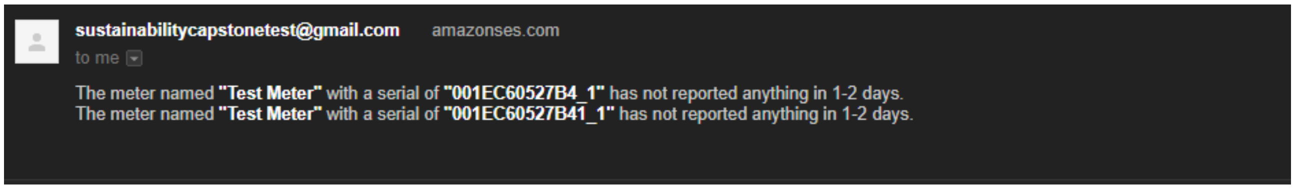
```
Meter.find().then(meters => {
        meters.forEach(e => {
            DataEntry.find({
                meter_id: e._id,
                timestamp: {
                    $gte: oneWeekAgo,
                    $lt: now
                }
            }, (err, docs) => {
                if (err) throw (err);
                // We want the length to be greater than 100 in the past week (96 are
                    entered per day)
                // Without this, a meter might report some weird things, so this forces
                    at least two days of entries
                if (docs.length && docs.length >= 100) {
                    pastData = docs.filter(t => {
                        return t.timestamp >= oneWeekAgo && t.timestamp <= yesterday
                    });

                    currentData = docs.filter(t => {
                        return t.timestamp >= yesterday && t.timestamp <= now
                    });

                    if (pastData.length) {
                        pastDataArray = [];
                        // gets the energy usage values for the past week
                        for (i = 0; i < pastData.length; i++) {
                            pastDataArray.push(pastData[i].point[0].value);
                        }
                        pastDataAvg = math.mean(pastDataArray);
                        pastDataSD = math.std(pastDataArray);
                        pastDataThreshhold = pastDataAvg + pastDataSD;
                        // checks for empty array so it doesn't report on down meter
                        if (currentData.length) {
                            currentDataArray = [];
                            for (i = 0; i < currentData.length; i++) {
                                currentDataArray.push(currentData[i].point[0].value);
                            }
                            currentDataAvg = math.mean(currentDataArray);
                            if (currentDataAvg > pastDataThreshhold) {
                                email.body += (`The meter named <b>"${e.name}"</b> with a
                                    serial of <b>"${e.meter_id}"</b> has reported high
                                   energy usage.` +
                                   ` Over the past week, the meter had an average of <b>
                                      ${pastDataAvg.toFixed(1)}</b> kWh,` +
                                   ` but has reported <b style="color:red">${
                                      currentDataAvg.toFixed(1)}</b> kWh in the past 24
                                       hours. <br>`)
                            }
                        }
                    }
                    if (meterCount++ == meters.length - 1 && email.body !== '') {
                        emailAlert(email);
                    }
```

3.3.3 *Down Meter*

Another requirement is to send out an alert email whenever a meter goes down. To do this, we check to see the last time each meter has checked in. If the last checkin was between 24 and 48 hours, it will send out a alert email. Like the usage function, this is only called once per day. Because of this, users will only ever receive one notification for down meters. This was intentional as it would be pure chaos if it sent an email every time data was received. The functionality of this function is shown below

```
function checkMeterTimestamps() {
    var yesterday = moment.utc().subtract(1, 'days').format('YYYY-MM-DD HH:mm:ss');
    var twoDaysAgo = moment.utc().subtract(2, 'days').format('YYYY-MM-DD HH:mm:ss');
    var meterCount = 0;
    var email = {
        body: '',
        subject: 'Meter(s) detected as offline!'
    }
    Meter.find().then(meters => {
        if (meters) {
            // finds data entries for each meter and checks the last data entries time.
            meters.forEach(e => {
                DataEntry.find({
                    meter_id: e._id,
                    timestamp: {
                        $gte: twoDaysAgo,
                        $lt: yesterday
                    }
                }, (err, docs) => {
                    if (!err) {
                        timestamps = docs.filter(t => {
                            return t.timestamp >= twoDaysAgo && t.timestamp <= yesterday
                        });
                        if (timestamps.length) {
                            email.body += `The meter named <b>"${e.name}"</b> with a
                                serial of <b>"${e.meter_id}"</b> has not reported
                                anything in 1-2 days. <br>`
                        }
                    }
                    if (meterCount++ == meters.length - 1 && email.body !== '') {
                        emailAlert(email);
                    }
                })
            })
        }
    })
}
```

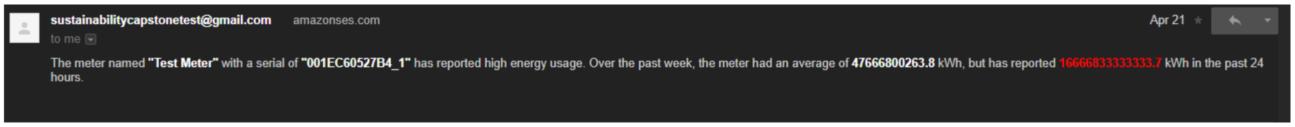
You will notice the email bodies of both the usage and down meter functions to have markup. The result is emails with formatted data. Below are screenshots of what each look like.



sustainabilitycapstonetest@gmail.com amazonses.com
to me

The meter named **"Test Meter"** with a serial of **"001EC60527B4_1"** has not reported anything in 1-2 days.
The meter named **"Test Meter"** with a serial of **"001EC60527B41_1"** has not reported anything in 1-2 days.

Fig. 4: A screen shot of an email showing a meter that is not reporting data.



sustainabilitycapstonetest@gmail.com amazonses.com Apr 21
to me

The meter named **"Test Meter"** with a serial of **"001EC60527B4_1"** has reported high energy usage. Over the past week, the meter had an average of **47666800263.8** kWh, but has reported 16666833333333.7 kWh in the past 24 hours.

Fig. 5: A screen shot of an email showing a meter that has detected energy usage spikes.

4 Contributor: Parker Bruni

4.1 Updates

4.1.1 Contact Page

The contact page has been recently finalized, as the requests for professional head shots of all members involved has been fulfilled. The contacts page now shows circular images associated with each members description on that page.

Requests were made by the head of the Office of Sustainability to exclude certain members of the Office of Sustainability and update the emailing functionality so as to reduce the issues related to email spam bots that may get access to our publicly facing interface. A general email has been used for members of the Office of Sustainability rather than personal emails to direct any inquiries about the office or the project to that location. Below is an example section of code that demonstrates how we display our members, as well as the system that we have implemented to reduce spam.

```
<div class="col-sm-4 p-0 h-100">
        <div class="container">
            <div class="w-100 pb-3">
                <img class="profile-pic border-orange" style="width:60%;" src="../../
                    assets/people/lety_square.jpg" alt="Card image cap">
            </div>
            <div class="w-100 p-2 ml-2">
                <p class="font-weight-normal" style="font-size: 150%; color: #DC4405;">
                    <a class="text-orange Stratum2-Thin">Leticia Cavazos</a>
                    <a href="http://fa.oregonstate.edu/email/node/936/field_group_email"
                        target="_top">
                        <img class="rounded" style="width:8%;" src="../../assets/email-
                            icon.png" alt="email">
                    </a>
                </p>
                <p class="Stratum2-Light font-weight-light" style="font-size: 110%;">
                    Program Specialist</p>
                <p style="font-size: 85%;">Leticia was hired as a student at the
                    Sustainability Office in June of 2014. In December 2016, she
                    transitioned
                    to the full-time position of Program Specialist. Her primary duties
                        include utility data acquisition,
                    management, and analysis, meter hardware management, and
                        sustainability assessment.</p>
            </div>
        </div>
    </div>
```

4.1.2 Public vs. Authorized User Access

Previously, public users that did not have an authorized user access account had the ability to exploit certain elements that existed on the public facing page to potentially manipulate the higher level access data. This would cause problems as a general user with malicious intent could delete buildings, public dashboards, public stories, and potentially more. This was possible because a UI element that had access to this functionality was being populated on public facing pages because it did not check the current users privilege levels. This element has since been updated to be hidden on public facing pages to prevent these issues. Below is an example of the code that controls if the UI elements are displayed or not.

```
<div ng-hide="userLoggedIn" class="float-right pr-3 dropdown" style="font-size:18px">
```

The variable ”userLoggedIn” is a simple switch statement variable that is controlled by the authentication implementation and checks if a user with correct privileges is logged in. You can see the module that controls this variable below.

```
angular.module('sideNavController', [])
    .controller('sideNavController', function ($scope, $rootScope, GetUser, Story) {

        $scope.loggedIn = true;


    });
```

5 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 Describe what you have left to do

- Complete meter subtraction for monitoring McNary residence hall consumption.
- Slight code polish. and then writing documentation.
- Writing documentation.

[1]

References

[1] Mongoose.js, “Sub docs.”