MEMCloud Documentation

Release 0.0.1

Ashray Manur

Jan 20, 2018

Contents

MEMCloud is the cloud platform for managing microgrids.

CHAPTER 1

Contents:

1.1 MEMCloud Architecture

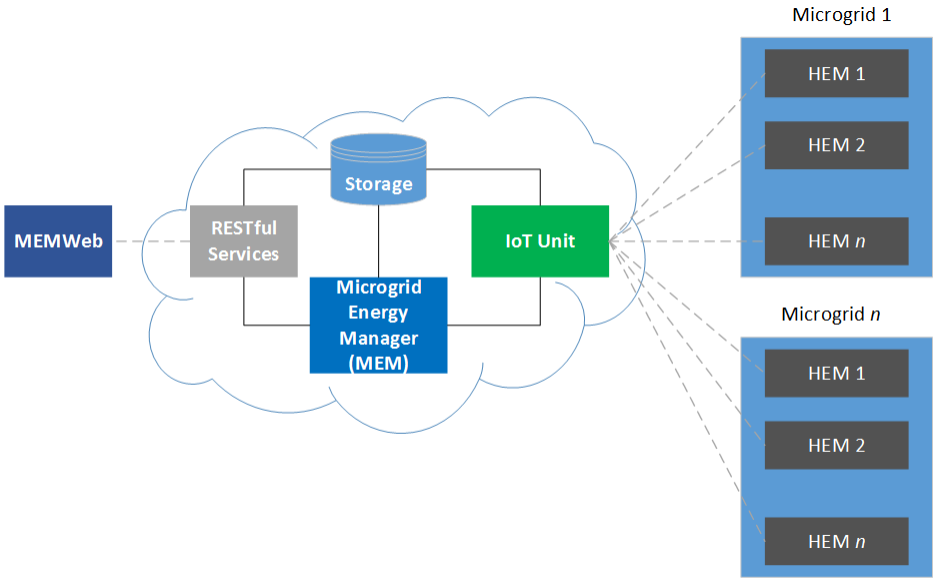
1.1.1 Goals

- Make it a platform based solution, allowing 3rd party developers to plug-and-play different services.
- Provide end-to-end capabilities so that users don't have to develop missing pieces
- Design suitable for energy systems (microgrids, smartgrids, building automation, IoT etc)

1.1.2 Architecture Overview

MEMCloud is a component of SEUP. Before you read about MEMCloud, check out the following resources to know more about SEUP and how MEMCloud fits into the SEUP puzzle

- SEUP Docs
- SEUP Publications



MEMCloud consists of five main components that include storage, RESTful Services, IoT Unit, Microgrid Energy Manager (MEM) and MEMWeb.

- RESTful service - serves the MEMWeb. It exposes a suite of REST APIs that MEMWeb can use to integrate with the backend services.
- IoT Unit - The IoT unit on MEMCloud acts as an interface between HEMCore and other components of MEMCloud
- MEM (Microgrid Energy Manager) - The MEM is the core of the backend infrastructure. It houses applications which are responsible for operation, management, control, and optimization of all the MGOs. These applications are developed by users and are deployed through MEMWeb

1.1.3 Setting up MEMCloud (MEMBackend, IoT Engine and DBs) on a local machine/server for testing and development

MEMCloud consists of two main components as far as code is concerned `MEM WebApp` and the code which corresponds to the `IoT Engine` and `MEMBackend`

Stack and Infrastructure details

- `Node.js` for both front-end(with `Express`) and back-end
- `Angular2` along with a host of other libraries for our Web applications
- `MySQL` for Web application related stuff.
- `MongoDB` for IoT related stuff.
- `RabbitMQ` for our messaging service
- `Docker` for containers

Setting up on your local machine

```
Fork a branch from production. This is the latest branch
```

- Once you clone the repository, `mem` is the main directory. Under mem, the distribution is as follows
  - `app` and `public` corresponds to the Web application/User Interface. The main file which is use to start the Web server is `app.js`. Note: The web app has environment dependencies at this point runs only on an AWS environment (it uses googleOAuth, AWS load balancer etc.). Work has to be done to configure to run on a local system.
  - Running the back-end which consists of all the IoT engine, MEMBackend and databases is relatively easier on local system
  - `cloud` is the main directory which has all files.
  - `clients` directory has the software clients (which emulate the hardware)
  - `package.json` is our `Node.js` dependencies.
- PART 1 - Installation and setting up of MEMBackend and IoT Engine
  - Setup a separate VM (maybe Ubuntu) in case things screw up. We run `Ubuntu server 14.04` on AWS.
  - Install `node.js` version `v6.9.x` from `https://nodejs.org/en/`. We use version `v6.9.2`
  - Install `MYSQL` You can use this guide (or any). We use version `5.5.44-0ubuntu0.14.0.4.1` - https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-14-04
  - Install `MongoDB` using this guide (or any) - https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-14-04. We use version 3.4.1
  - Install the latest version of `RabbitMQ` using sudo dpkg -i rabbitmq-server_<version number>_all.deb. You can use this https://www.rabbitmq.com/install-debian.html. We use version 3.6.5
  - Configure RabbitMQ replace `/etc/rabbitmq/rabbitmq.config` with `mem/cloud/rabbitmq.config`
  - To clear all exchanges and start the queues - use this `sudo rabbitmqctl stop_app && ``sudo rabbitmqctl reset && sudo rabbitmqctl start_app`
  - In mem/ directory, run `sudo npm install`. This installs all the `package.json` dependencies. Before starting the backend server, the log path has to be updated as per your local system in `mem/cloud/config.js`
  - Change line 4 : `var logPath = "./log/";` to whichever directory you want to collect logs in.
  - Import the MySQL and MongoDB schema (copy of both these databases will be provided). The mongoDB backup is mongo_backup_feb_3_2017. To use this as your database, you can run `mongorestore mongo_backup_feb_3_201``7. The mysql file is ``mysql_backup_feb_3_2017` in the mysql directory.
- PART2 - Creating and adding keys
  - Create a new directory keys in mem/cloud/
  - We need to create a self-signed certificate for TLS.
    - In `mem/cloud/keys`, execute `openssl req -x509 -newkey rsa:4096 -nodes -keyout private-key.pem -out public-cert.pem -days 365`
    - This will ask for a bunch of information. You can enter anything arbitrary (for testing and local development).

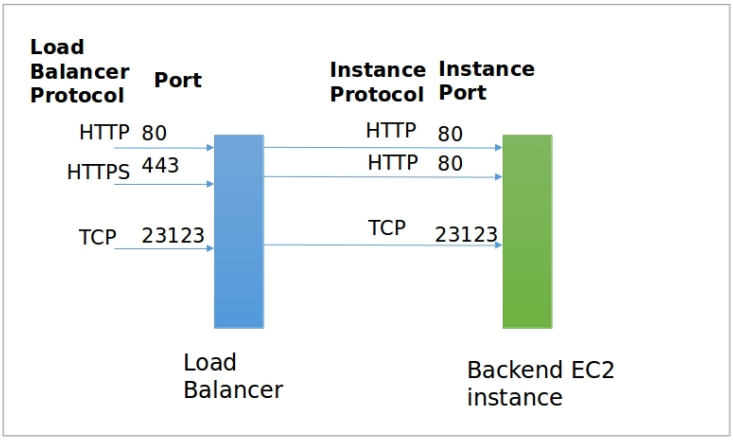
    * It then generates two files `public-cert.pem` and `private-key.pem`.
    * Copy `public-cert.pem` to `mem/clients/python`

- PART3 - Running server and client
  - Uncomment `line 41` (console.log) in `cloud.js` in `mem/cloud`. This will show the data the cloud is receiving.
  - To run the server, execute `sudo node run.js` in the `mem/cloud` directory
  - The console show the data received by the cloud (Hold on! You have to start the clients. See next step).
  - Before you start the clients, comment line 21 `(T="memcloud.io",23123)` and uncomment line 20 (`T="localhost",23123`). This ensures your clients connects to the local server that is running on your machine.
  - Now start the clients. The clients (which emulate actual hardware) can be found in `mem/clients/python`.
  - To start the clients run `python clients_tls.py 4` in `mem/clients/python`. This starts 4 clients.
  - The console shows the data the client is sending.
- Issues you might face
  - Installing Mongodb 3.4 on 32 bit ubuntu : Refer below link https://docs.mongodb.com/manual/tutorial/install-mongodb-on-linux/
  - While running server, you might get an error `Cannot find module bunyan & amqplib/callback_api`. To resolve this :In `mem/` run `sudo npm install bunyan;sudo npm install amqplib`
  - Change logpath from `./log` to appropriate path in `client_tls.py` file
  - In `databaseService.js` :Change the username and password to match the one you used to when creating mysql

1.2 Production Server Details

- Domain: memcloud.io, service provider - GoDaddy
- Admin email: admin@memcloud.io, service provider - GoDaddy
- Server hosting: Amazon EC2, service provider - AWS
- Load Balancing: Amazon Elastic Load Balancing (ELB), service provider - AWS
- Storage: 60GB Amazon Elastic Block Storage (EBS), service provider - AWS
- SSL/TLS for web applications: AWS Certificate Manager, service provider - AWS

1.2.1 Load Balancer Implementation



1.3 Directory Structure

This is the directory structure for the entire MEMCloud repository

In the `mem` directory

- `app` - this contains all the files for the web applications
  - `models`- the html and typescript for the front-end
    - `dashboard`- this contains all the typescript stuff. See *Web dashboard file structure* for more details about the web dashboard files.
    - `*.html` - all the html files for displaying static content
  - `appConstant.js` - constants for the HTTP related services
  - `appDatabaseService.js` - This should be deleted. This is the obsolete version of `webDatabaseService.js`.
  - `routes.js` - this handles all the routes for the web application. Pretty straightforward.
  - `webDatabaseService.js` - This acts as an interface between the RESTful service and database. Any request to pull certain info from the database is routed through this.
- `clients` - files for emulating the HEM clients
- `cloud` - contains all the files for the MEM
- `config` - contains configuration file for the web app
- `log` - contains log for the web app
- `modules` - this is dummy for now.

- `public` - contains all the public facing files - stylesheets, javascript modules, html etc.
- `app.js` - this file starts the web app
- `package.json` - all the JS module dependencies

1.4 Back-end File Structure

1.4.1 IoT Unit

- Responsible for sending and receiving data to/from HEM
- Also performs authentication and is responsible for encryption and decryption of data

Files

- `cloud.js`
- `commandProcessor.js`
- `commandSender.js`
- `msgTable.js`
- `socketTable.js`
- `msgPacket.js`

1.4.2 Chief Microgrid Operator (CMO)

- Creates one supervisor (`microgrid.js`) for each microgrid. Supervisor for a particular microgrid is created when the first HEM of that microgrid sends data.
- The CMO for now is `microgrids.js`

Files

- `microgrids.js`

1.4.3 Microgrid Supervisor

- Creates a message bus for a microgrid so that worker nodes can subscribe to it and get live data.
- When a job comes to the supervisor, it creates a worker node and asks it to run.
- For now, this is `microgrid.js` in `mem/cloud/microgrid`

Files

- `microgrid.js`

1.4.4 Status Monitor (SM)

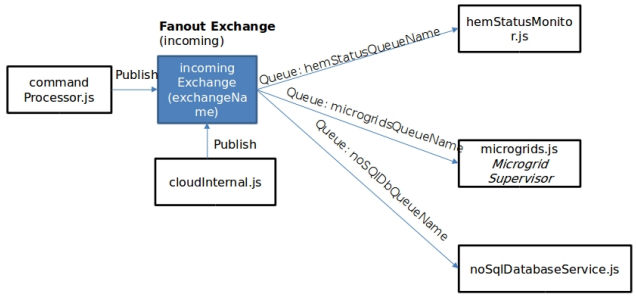
- Maintains status of all HEMs for the REST services. Mainly for the web application

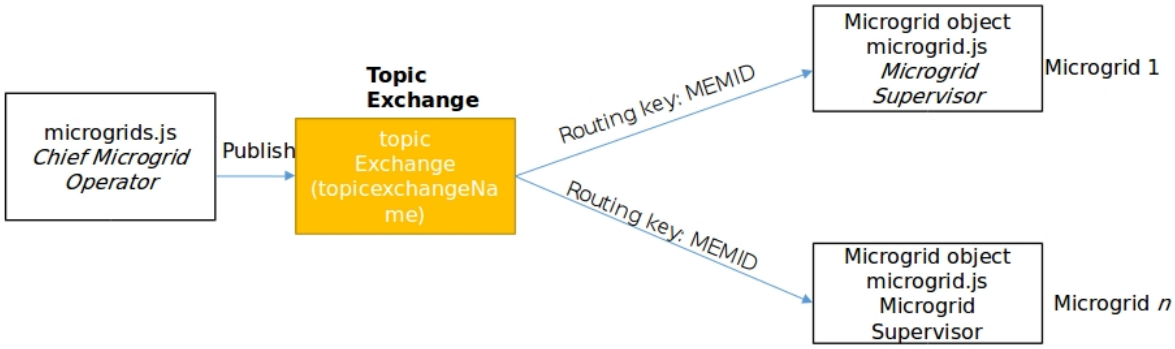
Files

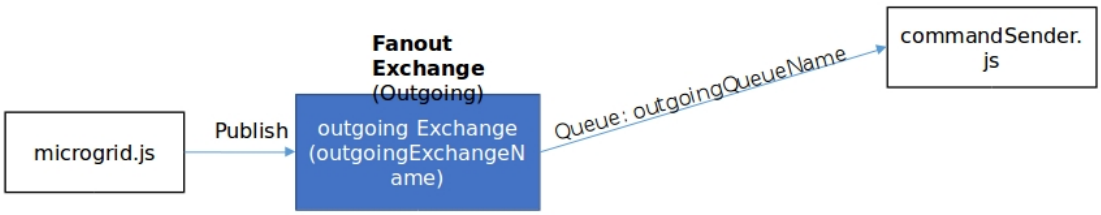
- `hemGlobalEvents.js`
- `hemStatusMonitor.js`
- `hemStatusObj.js`

1.4.5 Exchanges and Queues

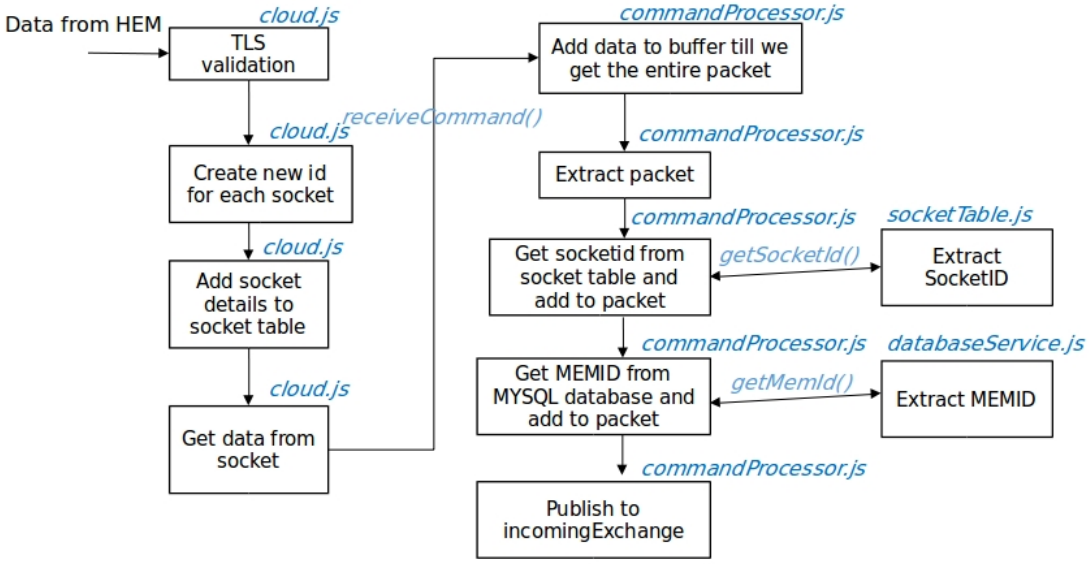
- We have two types of exchanges– fanout and topic
- Fanout distributes messages to all connected queues. It does dummy broadcasting
- Topic distributes messages selectively based on routing key

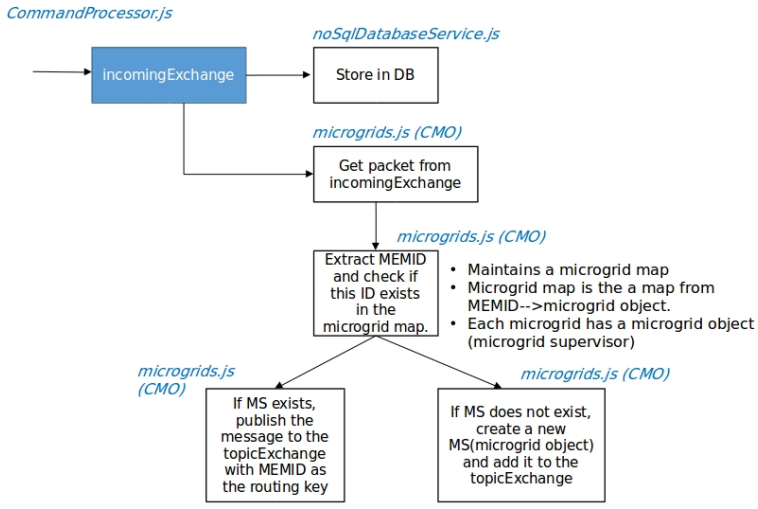


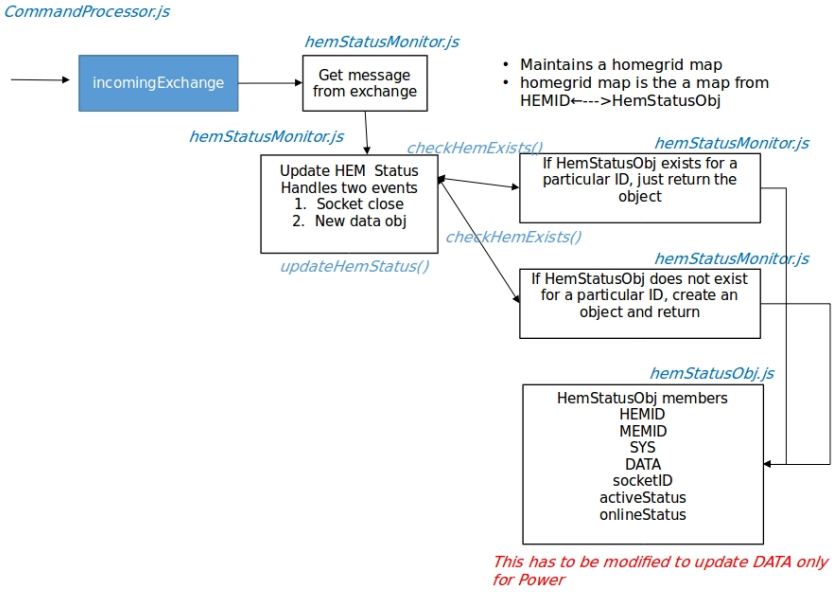

microgrids.js
Chief Microgrid Operator
Publish
Topic Exchange
topic Exchange (topicexchangeName)
Routing key: MEMID
Routing key: MEMID
Microgrid object microgrid.js Microgrid Supervisor
Microgrid 1
Microgrid object microgrid.js Microgrid Supervisor
Microgrid n

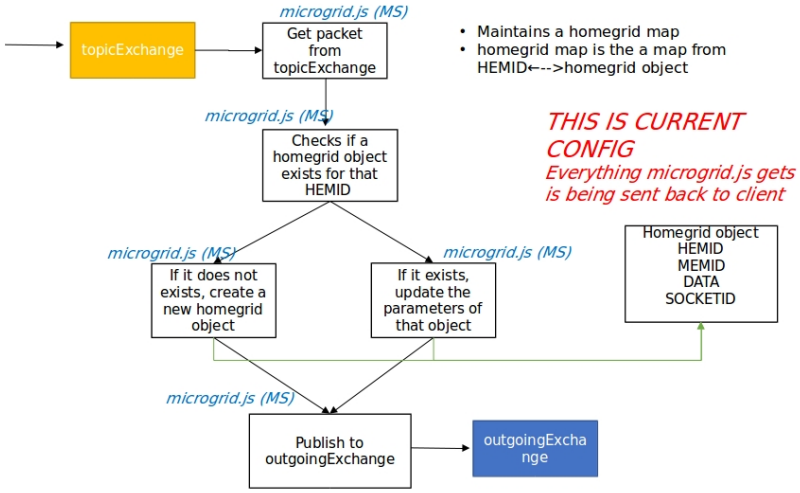

Fanout
Exchange
(Outgoing)
microgrid.js
Publish
outgoing Exchange
(outgoingExchangeN
ame)
Queue: outgoingQueueName
commandSender.
js

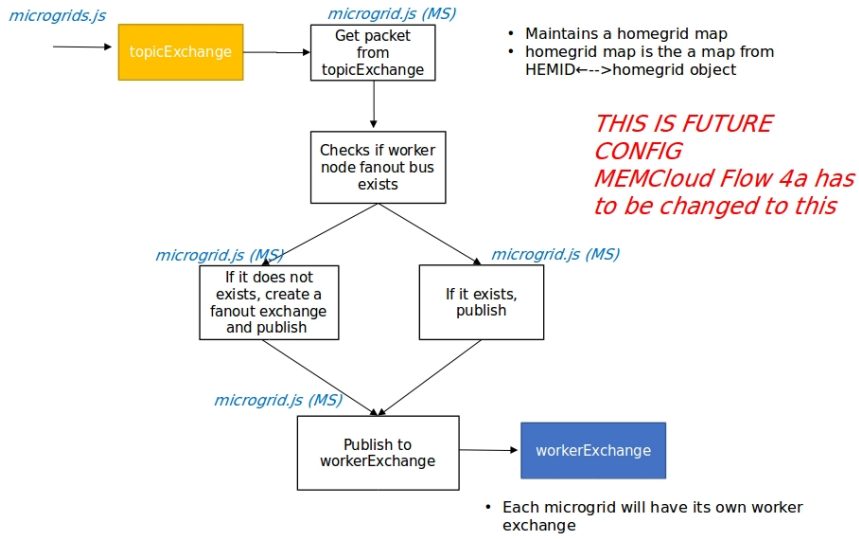
1.4.6 Backend Flow






CommandProcessor.js
hemStatusMonitor.js
incomingExchange
Get message from exchange
• Maintains a homegrid map
• homegrid map is the a map from HEMID<---->HemStatusObj
hemStatusMonitor.js
checkHemExists()
hemStatusMonitor.js
Update HEM Status
Handles two events
1. Socket close
2. New data obj
If HemStatusObj exists for a particular ID, just return the object
checkHemExists()
hemStatusMonitor.js
updateHemStatus()
If HemStatusObj does not exist for a particular ID, create an object and return
hemStatusObj.js
HemStatusObj members
HEMID
MEMID
SYS
DATA
socketID
activeStatus
onlineStatus
This has to be modified to update DATA only for Power


microgrid.js (MS)
topicExchange
Get packet from topicExchange
• Maintains a homegrid map
• homegrid map is the a map from HEMID<---->homegrid object
microgrid.js (MS)
Checks if a homegrid object exists for that HEMID
THIS IS CURRENT CONFIG
Everything microgrid.js gets is being sent back to client
microgrid.js (MS)
microgrid.js (MS)
If it does not exists, create a new homegrid object
If it exists, update the parameters of that object
Homegrid object
HEMID
MEMID
DATA
SOCKETID
microgrid.js (MS)
Publish to outgoingExchange
outgoingExcha nge



1.5 Front-end File Structure

1.5.1 Web dashboard file structure

- `gulpfile.js` - this contains all the info on how to compile typescript, how to run the app etc.
- `tsconfig.json` - this is the config file for gulp
- `process/typescript` - this contains all the typescript files for Angular
  - `main-dash.module.ts` - this is the root module which tells Angular how to assemble the application. Refer - https://angular.io/guide/bootstrapping
  - `main-dash.component.ts` - this is the root component of what will become a tree of nested components.
  - `main-routing.module.ts` - this module takes care of navigating from one view to another. This is the main routing module for the entire application. Refer https://angular.io/guide/router#milestone-2-routing-module

Parent and Child Hierarchy

- `main-dash.module.ts`
- `main-routing.module.ts`
- `main-dash.component.ts`
  - `modal.component.ts` and `modal-http.service`
    - `hem-list.component.ts` and `hem-list.service`
      - `hem-single.component.ts`

1.5.2 How does it work?

- As soon as the user logins in, `dash.html` is called which in-turn calls `<main-dash></main-dash>`. This triggers the creation of `main-dash.component.ts` module.
- This module is then created and calls `modal.component.ts` which then brings up the modal if the user has not been validated through a MEMID. It checks if the user has validated through `preModalCheck()`. If not, it brings up a form(modal) through `modal.html` in `mem/public/dash/partials`. The validation and submission of the form happens through `preModalCheck()` and `validateForm()`. It uses the service `modal-http.service` to communicate with the REST server. If the ID is valid and everything is okay, it brings up `<hem-list></hem-list>` which is in the `<div>` with the id `show`. If things dont work out, all the html belonging to that `<div` is hidden.
- Calling `<hem-list></hem-list>` created a new component `hem-list.component.ts`. The HTML associated with this is `dash/partials/hemlist.html`. The service associated with this is `hem-list.service.ts`. This service is used to get all the information for all HEMs. The component gets the data from HEM every few seconds using the service and then displays that data. It tells you the HEM ID, active status and online status of each HEM‘‘
- When you click on one of the HEMs, `navigateHem()` from `hem-list.component` is called which then passes that info to the router.
- `hem-viewControl.service` - how does that work? Refer to `hem-list.component.ts` and `hem-list.service.ts`.
- The `main-routing.module.ts` then matches this route and in this case, creates a new component `hem-single.component`. The router matches that URL to the route path s and displays the component after a RouterOutlet that you've placed in the host view's HTML. In this case that is `hem-list.html`.Refer https://angular.io/guide/router#router-outlet, https://stackoverflow.com/questions/40476814/angular2-target-specific-router-outlet/40477582#40477582
- For this particular route, `hem-single.component.ts` is created.

Angular Resources

- Angular quickstart - https://angular.io/guide/quickstart