SensorNet Web API

Background:

- System is made up of one root (RPi), and 1..many nodes
  - Identified by a 4-byte ID
- Each node has 1..many modules
  - Identified by a 5-byte ID (a concatenation of NodeID and the *first* Command – *Note: ONLY module IDs are unique to the system.* Ex. NodeID: 0xE7E7E7E7, Commands: 0x10 and 0x11, ModuleID: 0x E7E7E7E710
  - Modules are collections of related sensors and/or controls. Ex. Temperature sensor and thermostat
- Each module has 1..many sensors and/or controls
  - Identified by a 1-byte Command – *Note, first Command is always 0x10*
    - The command *is* the number. Sending a 0x10 to a sensor automatically retrieves data. Sending a 0x11 (and data with it) to a control automatically updates with that data
  - A sensor returns data. Ex. (1)char = 22 means 22°C
  - A[n] [automation] control receives data. Ex. Send (1)char = 0x01 means turn on

Data:

- nodeStatus: (0) all good (1) warning (2) critical (3) no communication. *Note: statuses "ripple" up, so that a sensor status > 0 will cause module status and node status to become > 0. However, a module status of 3 (no communication) will not cause sensor statuses to become > 0*
- moduleType: (0) sensor (1) control (2) both
- icon: WILL NOT BE IMPLEMENTED
- updateInterval: time to next update (in HH:MM:SS format)
- timestamp: time of last update
- moduleFile: name of file that contains client side processing for web app. Ex. Converting 24 bit value to HSL color (0x 666633 to HSL(60,90,21), aka goose poop green) *HINT! Much more intuitive to make the color picker HSL, then convert to RGB*
- commands: unique (within node) identifier of sensor/control
- dataLens: data length in bytes, usually between 1 and 30
- types: I can't remember… so don't bother using it
- archiveData: list of timestamp:value pairs of prev. data. Useful for graphing
- criticalBound and warningBounds: bounds of "safe" data. ONLY for sensors – controls will have NULL values. Ex. Temperature sensor may have warning bounds of [18, 26], and critical bounds of [15, 30] (unless you're measuring how hot I am, then there is no upper limit)

/getModuleList

Method: GET

Returns data intended to create a list of available modules

Returns:

JSON (list – *not* key-value pairs): {NodeID, ModuleID, Location, Name, Icon}

Code:

```
statement = ('SELECT Node.ID, Node.ModuleID, NodeDetails.Location, ModuleDetails.Name,
ModuleDetails.Icon \n'
        'FROM Node\n'
        'INNER JOIN NodeDetails\n'
        'ON Node.ID=NodeDetails.ID\n'
        'INNER JOIN ModuleDetails\n'
        'ON Node.ModuleID=ModuleDetails.ModuleID\n'
        'GROUP BY Node.moduleID;')
```

/getStatsList

Method: GET

Returns general statistics about the system: # of nodes, # of modules, # of sensors, # of controls, # or warnings, # of criticals

Returns:

JSON (list): {nodes, modules, sensors, controls, warnings, criticals

/getModuleInfo

Method: GET

Sent: ModuleID

Returns all information about requested module in JSON. All info are in key:value pairs. All values with empty [] are lists – for each module, there may be > 1 sensor and/or control, so all values with empty [] means that there may be multiple values returned, one for each command. If querySuccess returns false, then (most likely) there was a problem communicating with the node.

Code:

```
moduleData = {
                    'querySuccess': True,
                    'moduleID': None,
                    'ID': None,
                    'nodeStatus': None,
                    'name': None,
                    'description': None,
                    'moduleType': None,
                    'icon': None,
                    'updateInterval': None,
                    'timestamp': None,
                    'moduleStatus': None,
                    'moduleFile': None,
                    'commands': [],
                    'dataLens': [],
                    'values': [],
                    'types': [],
                    'status': [],
                    'archiveData': [],
                    'criticalBounds': [],
                    'warningBounds': []
                    }
```

/updateControl

Method: POST

Send JSON list: [‘updateControl’, moduleID, [commands], [values]], where ‘updateControl’ is a literal string, for protection (tried a condom on the antenna, didn’t work), and commands and values are lists. *NOTE! Only sensors/controls in that (one) particular module can be updated at once.*

Returns: nothing (yet?)

/updateModule

Method: POST

Send JSON list of key:value pairs, similar to list in /getModuleInfo, but only sending moduleID, name, description, updateInterval, criticalBounds and warningBounds. These are the only things you can change about a module. UpdateInterval is intended to stay constant, which may mean that upon updating, sensor may be updated sooner than expected. Ex. Timestamp was 5 minutes ago, and updateInterval is set to 10 minutes. Setting updateInterval for 15 minutes will cause sensor to be updated 10 minutes from *now* (when the new update interval was set) It’s a “bug” that I don’t have time to fix, however there is a workaround below…

/refreshModule

Method: GET

Send: moduleID in JSON: [‘refresh’, moduleID]

Returns: JSON: [moduleID, [commands], [values]]

Workaround to above “bug” is to refresh immediately before changing updateInterval, therefore updating timestamp to *now*