Brief Developer Summary of Green Home Web App:

Introduction:
Backbone.js was the framework used. If you are unfamiliar with it, read over these docs http://backbonejs.org/ and study the annotated source code of this 'Todos' app (http://documentcloud.github.com/backbone/examples/todos/index.html) – as many of the same techniques are used in my code.

Flot.js is an amazing graphing library that has only Jquery as a dependency. All of the graphs are rendered using the 'time' setting (as opposed to sequentially plotting statuses in rigid fashion), meaning that all time series must have timestamps (as millisecond UTC integers) as on of the attributes in the statuses of the series.

General javascript tips: Develop in Chrome, don't settle for less. Using console.log and console.debug are enormous timesavers, learn to use them well! You will come to love JS!

Personally, I love the Backbone framework. Its models/collections keep Javascript's 'callback hell' and scope issues in check.

--For specific questions, email me at sdailey.479@gmail.com - Spencer Dailey

Summary of the purpose of each JS source file:

ApiCommands.js –

- Study the contents of this file! The command_center(command, params) function is *THE* centralized way we GET /POST from/to the GreenHome Server Api – making ApiCommands.js the lowest level of this webapp's 'network stack'. (Below you'll read about the getAssembler.js file, which parses complex callback objects from the server – but some of that might not 'click' if you overlook this file.)

appView.js –

- instantiates initial views ('SessionView' for example),
- checks to see if the domain's cookie is valid, will load the app's previous state from the userAppModel if that is the case (see userModel.js)
- includes a glossary of global events

SessionView.js –

- Handles the initial setting of the cookie
- Gets the initial device information, creates model for each Device, adds to "Devices" collection (getting via ApiCommands.js)
- handles the visual rendering/animation of the Login/logout button

userModel.js –

- Creates a persistent Backbone model which gets added to the 'savedUser' Collection that -- Holds all of the app's 'state' information (such as what panels are in what locations when the app closes)
- You'll see throughout the app the following references to this model
    - 'userAppModel.set(…' , 'userAppModel.get(…'

Panel_Views.js-

- There are two different 'Views' in this file: 'PanelsContainerView' and its sub-view: 'PanelView'
- The purpose of the Panels Container is to listen to 'panel creation' events and render the appropriate panel sub-view.
- There are 4 different kinds of 'panels' (of Panel View):
    - the starter panel (the initial "Welcome" panel),
    - the device-specific 'datatype' graph (such as the kWh panel for a Microwave). These are created in groups with one for each supported graph type (an attribute in the userAppModel),
    - the 'custom panel' (admittedly a bad name for it) – which displays a stacked graph for all devices energy consumption,
    - the 'insights panel' – which displays device correlation information
- Because each panel type is so different, this file is large and has a whole lot going on. There is a lot of jquery object binding. If you find the code hard to follow, I would keep an eye out for the .bind(…, .click(…, or .on(… functions found at the beginning of each View

HomeDataView.js-

- Renders the 'widget area' in the bottom left part of the app and makes sure its in sync with the sidebar view.

getAssembler.js-

To render any stacked graph or interface object requiring multiple callbacks, USE THE GET ASSEMBLER, it will make your life easy! :D

get_assembler (successEventObject, series_descriptor, callbackEvent) {}

Here's a general description of the above function and how to use it:
The plan is to ask for a bunch of objects from the GreenHome server and then to receive them all at once.
We need to set certain parameters that define the objects we are requesting, we will add them all to a 'series_descriptor' argument as objects inside of an array.

Next… the "success trigger"

We pass a 'successEventObject' as an argument into the get_assembler() that serves as a trigger (that get_assembler() triggers when it's done fetching and parsing objects from the server and everything is ready). (the class SuccessObjectClass() is defined at the beginning of the file and is very small, so read it.)

When the get_assembler is done it "puts" its information in a private variable of this class instance, available when the object is triggered.

What's provided on completion (or the 'success trigger' event) is an array of callback objects in the same order requested, prepared as time series of 'statuses' (complete with timestamps for each status). If you would like to see the results directly, use console.debug(successobject_info) and look at the object in the chrome dev console.

The get_assembler also supports special requests as defined in the "SPECIAL DATATYPES CENTER" of getAssembler.js, such as 'houseEnergy' which sums all of the devices kWh values and returns a total sum.

Here is a code example for requesting from the get_assember seven days worth of values for a specific panel type:

(From panel_views.js):

```
seven_days: function(){
                    //DEFINE THE RANGE, DATATYPE AND FIDELITY OF DEFAULT PANEL GRAPH
                      var to_ = time_parser("DATE", "NOW", true);  //returns in UTC milliseconds
                      var from_ = time_parser("DATE", "HOURS-FROM-PRESENT", true, 168);  //returns in
UTC milliseconds

                      //DATA FIDELITY (LOWER NUMBER (of MINUTE INTERVAL) MEANS HIGHER FIDELITY)
                      var blocksize_ = 30; //our statuses will be for 30 minute intervals

                      var datatype_ = this.panelType;  //such as 'powerDraw' or 'energyConsumption'
                      var deviceId_ = this.deviceId;
                      var deviceName_ = this.deviceName;
                      var callbackEvent = 'sevendays';  // this merely needs to be a unique string for event
binding, nothing more

                      //Handle the success by instantiating new trigger 'object',
                                  //inside we put the code we wish to have executed after the
                                  get_assembler has //done its job
                      successObject = new SuccessObjectClass(); var callbackContext = this;
                      $(successObject).bind(callbackEvent, function(e, it){
                            var finalSeriesWithData = it.get_series();
                            callbackContext.graph_render(finalSeriesWithData['seriesArray'][0]);
                            });

                      //Describing the series we need for the graph
                      get_assembler(successObject, [{'from':from_,'to':to_, 'blocksize': blocksize_,
'datatype': datatype_, 'deviceId': deviceId_, 'deviceName': deviceName_}], callbackEvent);
   },
```

In the above example, there could be multiple objects within the [ ] brackets, we just happen to need one returned here. The text highlighted in blue is what gets executed upon success.

StatsView.js –

- Calculates and renders auto correlation and cross correlation between devices, using get_assembler very heavily

CurtainView.js

- Handles the rendering of the 'side curtain' images

Devices_Statuses_Models.js-

- Each Green Home device has an associated model. Read the attributes on these models as they are constantly being retrieved throughout the app.

Sidebar_Views.js-

- Listens to events in the app and changes/rerenders the sidebar html when the view is changed.