P.O.W.E.R. High Level Design Document

Grace De Geus
gdegeus@vtc.edu

Charles Hathaway
chathaway2@vtc.edu

Nate Pickett
npickett@vtc.edu

Niloc Quimby
nquimby@vtc.edu

November 7, 2012

Contents

List of Figures

1 Introduction

1.1 Purpose of this Document

This document describes the high level design of all components of the Power Outlet Wireless Reporter (POW-R) system.

1.2 Architectural Assumptions

It is assumed that an entire system consists of one Server, and one or more Satellites.

1.3 Summary of Requirements

The POW-R project aims to produce a low cost, low powered, device that will allow home owners to track their power consumption on a per-device basis. The hope is that by allowing home owners to see where all their power is going, they can take steps to disable or replace the most inefficient offenders.

To accomplish the stated purpose, this project provides the following components:

- Satellites
- The Server
- The Display

Each of these components have their own requirements. The basic summary of each one would be:

1.3.1 Satellite Requirements

- Ability to plug into standard NEMA 5-15 mains electrical outlets
- Two outlet sockets on the opposite side of the plugs (if it takes up two outlets, it should provide two outlets)
- Less than 5% error on current and voltage reading
- Power draw less than 1W (Watt) per Satellite
- Broadcasts information up to 500m (meters)
- Transmits every 1.0 seconds
- Zigbee compatible
- Encrypted communication

1.3.2 Server Requirements

- Physical Server inside the monitored building
- Runs on less than 10W (Watts)
- Hosts web server for Display
- Button to connect Satellites
- Ability to connect to the user's network
- Zigbee compatible
- Encrypted communication

1.3.3 Display Requirements

- User Management
  - Add users
  - Delete users
  - Modify user information
  - Modify user permissions
- Group Management*
  - Add groups
  - Delete groups
  - Modify groups

  - Group permissions
- Device Management
  - Add Devices
  - Disable Devices
    * Does not include deleting Devices, since that would invalidate history
  - Modify Devices
    * Rename
    * Change outlet association
  - Device Groups*
    * Device groups are logical groupings of Devices that can be viewed as a single Device in the View Data section
    * Create Device groups
    * Delete Device groups
    * Modify Device groups
- Satellite Management
  - Add Satellites
  - Remove Satellite
- View Data
  - View Device Power Consumption
    * Shows the user power consumption on a per-device (or device group) basis
  - View Power Consumption Over Time
    * Shows the user power consumption over a specified range of time
    * Includes total and per-device power consumption
  - View Device Cost Over Time*
    * Shows the user the cost to run a device, or group of devices, over a specified time range
    * Shows the user a comparison of device costs over a specified range with a given interval
    * E.g. Show them the device cost every month for the past year
- Power Bill Guesstimator*

  - View cost to run a specified device over a period of time in the future
  - View cost to run multiple devices over a period of time
  - View expected power bill if a device is added
  - Allow the user to specify cost of power for
    * Specific time ranges
    * Specific power-usage ranges
    * E.g. Power cost the user $0.08 per kW/hr for the first 500 kW/hrs, and $0.10 after that
- Factory reset*
  - Allow the user to reset the system to default settings via...
    * A form
    * A physical button located on the server
- Software upgrade*
  - Allow the user to upgrade the system to the newest software version

2 Hardware Design Overview

2.1 Description

The POW-R project is comprised of two main hardware components, the Server and the Satellites. The Server refers to the physical hardware from which the Display shall be served. It also acts as the data center for all Satellites associated to it, collecting data and administrating. The Satellites will talk over ZigBee specification to the one Zigbee module connected to the Server. That one Zigbee module can talk to the Server over Universal Serial Bus (USB).

Figure 2.1 shows the layout of the hardware system.

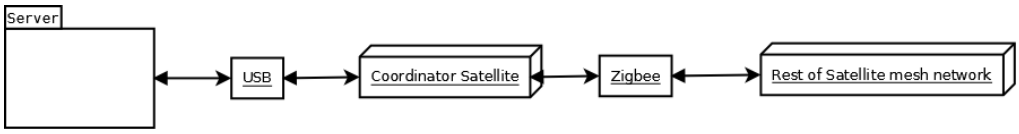

Figure 2.1: System Overview

2.2 The Server

2.2.1 PC Hardware

Mainboard

The Server's main hardware component is the mainboard, a SYS9400-ECX Developer-Ready Reference Platform. It's a small form-factor, low-power machine with the following specs:

- 1.6 GHz Intel Atom E6XX Series Processor
- 1 GB DDR2 RAM

- Roughly 6" by 4"
- Various connection interfaces:
  - 2x SATA ports
  - Header for Solid State Drive (SSD) power
  - Ethernet port
  - 5x USB 2.0 ports
  - General Purpose Input/Output (GPIO) pin header

Potential Problems

If for whatever reason using this mainboard falls through: It should be noted that the requirements for the Server hardware concern not just specifications, but interfaces as well. In particular, this project requires at least Ethernet, 2 USB ports, a SATA port, and accessible GPIO pins.

Storage

The Server's mainboard is connected via SATA to a 40GB SSD.

Power Supply

An adapter rated for 12VDC @ 3A is used to connect the Server's board to mains electricity.

2.2.2 Add-Ons

The Server provides for the computational needs of the POW-R project; More hardware shall be added before the utility needs of the project are met.

IP Display

The IP of the Server shall be displayed somewhere on it's casing. The user will enter this IP into their browser to access the Display.

This will be achieved by connecting an Arduino via USB to the Server, and housing it inside the Server's casing. The Arduino will be connected to an LCD display which will output the network IP of the Server. Figure 2.2 shows the interaction between Server and Arduino.

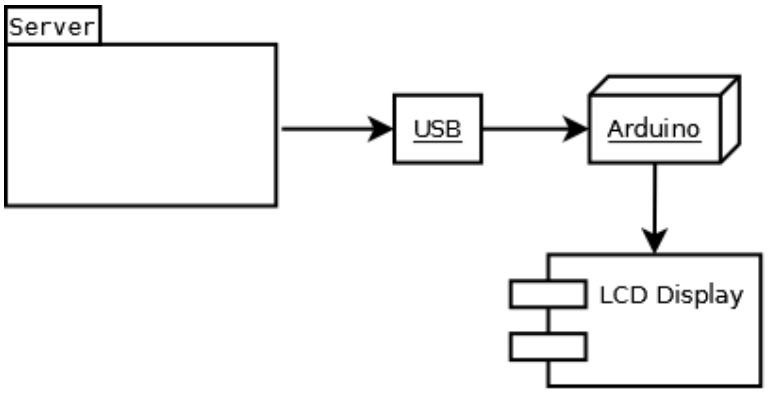

Figure 2.2: IP Display Diagram

The IP of the system can be obtained via kernel module, and sent to the Arduino one byte at a time. This works well, since the Arduino connects over serial and thusly takes a byte at a time.

2.2.3 GPIO Pins

As mentioned above in the "PC Hardware" section, the mainboard is outfitted with a GPIO pin header. Linux will be used for the Server's operating system, which means that Linux has to have a way to interact with such pins.

A folder can be found in the Linux kernel, at the location `/sys/class/gpio` that helps with GPIO manipulation. To set up a single GPIO pin, one must type the following command into a terminal (as root):

```
$ echo N > /sys/class/gpio/export
(where N must be a GPIO pin number)
```

When this command is issued, a directory is made inside the `gpio` directory, named `gpioN`, where N is the GPIO pin number you passed. Two files will be in that new directory, `direction` and `value`. `direction` can only contain "in" or "out" with no leading characters, spaces, line breaks, etc. `value` can only contain "1" or "0" and can be read or written to intermittently. These files, `direction` and `value` are responsible for what kind of pin it is (input or output) and what the current value is (1 or 0), respectively.

NOTE: The pin number you must echo into `/sys/class/gpio/export` is not necessarily the number of the actual pin on the board, but may refer to the pin of the bridge that connects the GPIO port to the motherboard.

The subsections below are buttons that the Server must have, and each of these buttons connects to a GPIO pin.

Power Switch

A standard rocker switch will be added to the case to provide the user with a way to turn the Server on and off. The style of the switch will clearly signify "On" or "Off".

Factory Reset*

This button should intentionally be placed somewhere inconvenient: Sunken into the case far enough that you need a skinny rod (such as a paperclip) to push it, and in a spot that chaotic forces (children, mean people, God's divine will) will not notice it.

Connect to Satellite

A button will be added to the Server that allows a user to add a Satellite easily. When a user wants to add a Satellite to the network, the following series of events should take place, in order of appearance:

- User plugs Satellite into wall outlet
- User presses "connect" button on Satellite
- User presses "connect" button on Server
- Satellite is now connected

2.2.4 Server Hull

The Server needs a protective casing, for several reasons. On the physical level, a durable casing will protect the hardware. The casing also serves to render unnecessary ports inaccessible to users. Lastly, the casing is important in the sense that the term "Server" currently only applies to the mainboard, and most people think of a legitimate Server as something physical, encased in a box, that's protected and hidden away, exactly as a Server should be.

As far as prototyping goes, the casing can be as simple as a folded piece of aluminum with holes cut in it to fit the IP display, the buttons, and any ports that must be exposed. As an end-game product, the casing would probably be a little less "junkyard."

2.3 The Satellites

The Satellites are made up of two sets of hardware: Instruments for measuring current and voltage, and radios for communicating that data to the Server. The end product shall have a hard casing around it, NEMA 5-15 sockets on one side, and prongs for a NEMA 5-15 male end on the other. However, it should be noted that prototyping will most likely involve all of the hardware being spread out on breadboards.

2.3.1 Measurement Hardware

2.3.2 Communications Hardware

XBee Series 2 Radio Modules

Interface Board

2.3.3 ZigBee Standard

ZigBee is a standard that uses 802.15.4 wireless protocol as it's foundation. It's particularly useful for setting up ad-hoc mesh networks. Much like how any Bluetooth devices can talk to each other, any ZigBee devices nearby can talk to each other (so long as they have the proper permissions).

Why ZigBee?

The advantage of using ZigBee is that it lines up with the requirements of the POW-R project. It's a specification targeted for low-power, low-data applications that need to be secure.

Another advantage of using ZigBee is that XBee radios, which also fit the requirements of the POW-R project, are ZigBee-compliant, so the forming of a mesh network can happen quickly and easily.

Device Types

The nodes of a ZigBee network are obviously ZigBee devices, but from the network's perspective, there are three distinct types of ZigBee device:

- Coordinator
- Router
- End Device

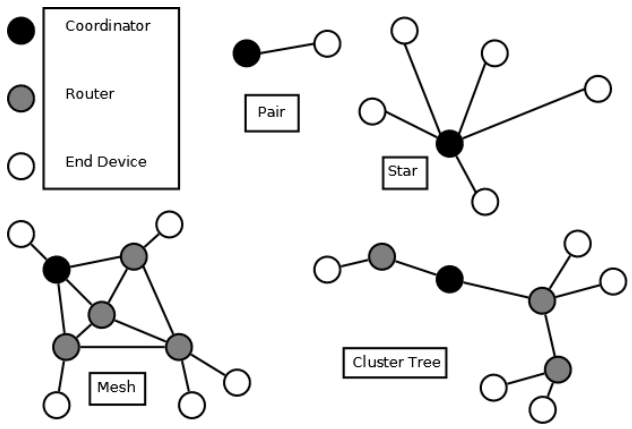

Figure 2.3: ZigBee Network Topologies

The coordinator runs the network, keeps it healthy, makes sure everybody is talking when they should be, and overall, just maintains the Mesh. It is also the device that is typically plugged into a computer via USB or some other serial interface, meaning that this is the node you want to send data to. There cannot be a ZigBee network without exactly one coordinator.

Routers are fully functional ZigBee modules, capable of taking care of themselves and routing messages across the Mesh. Routers are capable of being "parent" devices.

End devices are low-power ZigBee devices that don't do any routing. They can only talk to one other device, which is referred to as it's "parent" device. A router or a coordinator may be a parent device, but not another end device.

In the POW-R project, the coordinator will be attached to the Server so that it may also act as a data bridge between the Server and the Satellites. The rest of the Satellites are routers; There will be no end devices in the POW-R's mesh network.

Network Topology

Zigbee supports four network topologies, shown in Figure 2.3. A ZigBee network is defined by two rules:

- A ZigBee network must have two or more ZigBee devices
- One of those devices *must* be a coordinator

As mentioned before, the POW-R project will use the Mesh configuration, but without end devices. This means that the Mesh will consist entirely of routers and one coordinator.

3 Software Design Overview

3.1 Description

The POWER project uses various pieces of software to control the Satellites, provide an API for displays, and generate pages for the primary display. The POWER project will consist of the following pieces of software:

- The microprocessor code
- The server code to communicate with the Satellites
- The server code to present the API
- The client code to render the display for the user

The microprocessor code is documented in the hardware section because it is very related to the hardware, and it is much more logical to put the code there.

3.1.1 The Display Backend

The general plan for the backend is to write modules for a Django web application. These modules will either be running in the background and reporting the data given to the server from the Satellites to the database through the REST API functions, or responding to user requests sent through the REST API.

The logic for using Django as the base, and write modules around Django, is one of necessity. We do not have enough time or manpower to make everything we need from scratch. Therefore, we will be using several open source utilities to assist us.

Django will give us an easy-to-use database interface. It will deal with sanitizing all SQL queries, and make sure all output is escaped (ie, no XSS). Using Django gives us almost all of the security benefits listed in our non-functional requirements.

To make our REST API, we will utilize an open-source Django library called tastypie. Tastypie gives us both model resources (that is, api elements that are basically the database tables) and the ability to create custom resources that may, or may not, map to the database. The one (that's right, one) instance where we may not want to directly map something to the database is when we are looking at turning the outlets on and off.

Another advantage of Django is that is comes with a very extensive, expandable, and adaptable authentication system. This will help accomplish several optional and required features almost instantly.

3.1.2 The Display Frontend

The general plan for the frontend is to serve up some fairly static pages, and use extensive amounts of Javascript to fill in the content and animate things. The frontend will communicate with the backend through the REST API. This will be fairly simply, since the backend API will support multiple formats, including JSON.

To assist us on the frontend, we will be using several open source libraries. They will all serve a very clear function.

- Backbone.js - A library to provide basic functions dealing with fetching and syncing data with the server
  - Backbone.js will deal with tracking the "model" portion of our Javascript code.
  - Backbone.js will be made compatible with our REST API by using a small extension called backbone-tastypie
- iCanHaz - A library that provides advanced templating functionality. This is client-side templating, and relies on several "fake" HTML tags
- RequireJS - A library that deal with tracking and loading assets asynchronously. This includes loading libraries that we will be using
- jQuery - jQuery will be used for both parts of the front end and for the ajax support
  - We will use jqplot for dealing with graphs
  - We will use the jquery UI to deal with most of our widgets and such

3.2 Program Flow

This program will have three main "threads". One thread would be responsible for listening to messages from the Satellites, and another would be responsible

for listening and responding to requests on the API. The last thread would be responsible for serving up static files for the main display.

The Satellite thread will be responsible for taking the logging information provided by the Satellites and storing it in the database. Each Satellite will report it's information to the server once every second. Therefore, all this thread needs to do is react to incoming connections.

The API thread would provide the Representation State Transfer API (REST) that will be used by all future displays to get data in a standardized, well defined way. This thread will NOT be serving up images or page templates. This is done to make best use of the caching functionality included in the HTTP server software that will be used.

The last thread will be running a "dumb" web page that servers up static HTML and Javascript files. This thread will also retrieve images and CSS files from the server.

3.3 Data Flow

This application will use a SQL database to store all data. Each module will interact with the REST API, which interacts with the database as needed. The diagram for this database is broken down in the "Sub-modules" section, with a diagram for each logical grouping of tables.

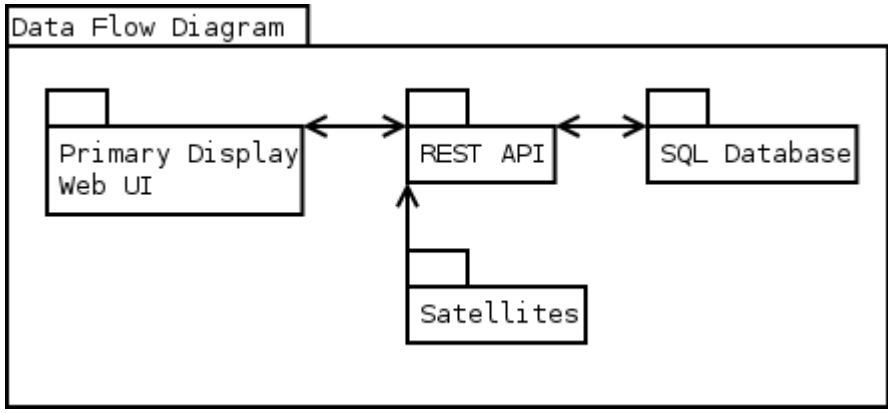

Figure 3.1: Data Flow Diagram

Figure 3.1 shows how various components will be interacting. Notice that the database is not directly accessed by any components, but goes through the REST API instead. This allows us to provide a standard interface should be

more-or-less accessible from any programming language or system. This will also allow us to use some well-known security features, such as TLS.

3.4 UML Diagrams

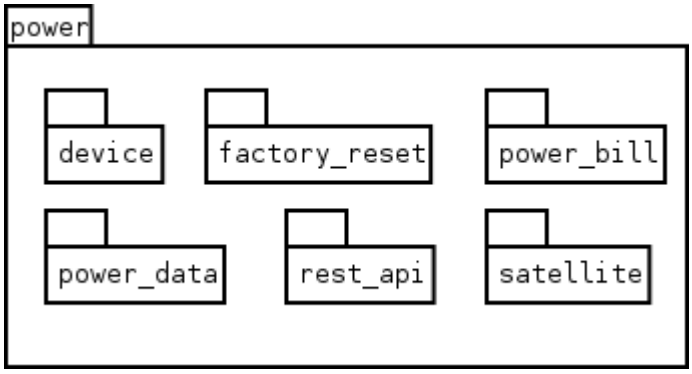

Figure 3.2: Listing of software modules

Figure 3.2 contains all the Django modules that will be created inside the actual Django "website" itself. This is basically the folder hierarchy of our web application source code. The "power" folder consists of folders for each module (not including the various open-source modules we are using, since those are being controlled via PIP). Each of these contains models.py file, which specifies what the database for that module should look like, a __init__.py file which will register the module with the REST API, and whatever other files that module needs.

3.5 Potential Problems

Using open-source libraries requires us to learn them. This is a big problem because of the time constraints on this project, and will require us to spend almost as much time learning as we will be in development (possibly more). This can lead to frustration, especially in light of the fact that we will need to learn the languages in addition to the libraries.

However, using well developed and designed libraries allows us to skip over a good part of the development and planning. We will be able to use extensive API's without taking responsibility for maintaining the software. This will allow us to focus on our product instead of the all the small things that make it up.

Another potential problem with using the suggested architecture will be clients that don't like using Javascript in their browser. Using heavy javascript allows us to offset a good bit of the load from our low-powered server to the clients computers, which will have a good deal more power. If we wanted to support clients that don't like using javascript (ie, security nuts), we would be looking at adding a significant load to our server (text-processing can consume a good bit of resources, especially string manipulation). Although we may be able to do, for the first version we will simply add "Javascript support" as a requirement for the clients.

3.6 The Backend

3.6.1 Authentication

Description

This module describes the architecture of how the backend will manage features related to users and groups (authentication features).

The backend authentication management provided by Django is very robust. It includes user authentication, access control (including groups and individual permissions), administrative panels, and logging of who does what.

Program Flow

The authentication module will be called whenever a user tries to access something. It will verify that the user is logged in, has permissions, and even write a short log message if it is an administrative function.

Data Flow

Because this module is not being written by us, it will not be following the standards we are using. Therefore, it will be using the standard Django classes to access database objects instead of going through the REST API.

While that is not ideal, it should be OK anyway. This part of the Django framework has been tested extensively, in many production environments. There is a low risk that something will not work, and if there is a problem there is a very high chance that upon being reported to the Django project, it will be fixed quickly.

Something worth noting is the way passwords will be handled. Django hashes all passwords in a variety of algorithms, as specified in the password field. It also salts the passwords. This is much better than the lazy way of hashing it

(maybe) and storing it in a database column. For more details, please see `https://docs.djangoproject.com/en/dev/topics/auth/` for more information.

Potential Problems

Like I stated above, the largest potentional problems will be bugs in this piece of the code. Because we did not create, and we will not be supporting it, any bugs that we find will need to be reported to the Django framework community and we will have to wait for them to roll out a fix.

Another problem that might be common is the difficulty in expanding on the user profiles. This will not be a problem for us because Django did such a good job in keeping their framework extensible. All we need to do is declare a OneToOneField in our model, and add a few panels to the administrative interface and we will have an instant user profile. Not only that, but we will have successfully separated user information from the authentication information, which is considered "good practice" as it is faster and more secure.

3.6.2 The REST API

Description

This module describes the architecture of how the backend will manage features related to providing access to the REST API.

The REST API will provide a urls.py file, which will provide a handler for all REST methods. These could be achieved by simply using the tastypie URL's, which can be generated by calling include(api_v1.urls) in the urlpatterns object.

Program Flow

The REST API will be activated by the Django chain when a request comes in for the access to the API, or when another module directly calls functions in the API during their response to a request from the user.

Data Flow

All data will be returned to the calling client either as a Python dictionary (if called directly from another module), or as JSON, XML, or YAML through a web request (as specified in the query filter using ?format=json).

The REST API will directly access the database. Initially, tastypie will be dealing with converting the HTTP request to a method on the Django objects.

Django will be generating the SQL that actually gets run, and encapsulating objects in Python classes.

Potential Problems

At first, we will be using a prepackaged Django module to help provide the REST API. This could be problematic because we don't know a lot about the implementation of the API, and if there is a problem tracking it down could be difficult.

The good news is that django-tastypie is well supported by the community, and source code is liberally commented. It follows Python commenting standards, and should be fairly easy to navigate. In addition, members of our team have already used tastypie for projects in both work and the garage.

The most problematic bit about using tastypie for the initial run-through is the compatibility problems it has with Backbone.js. However, the community has developed an addition to Backbone to help deal with this. It is called Backbone-tastypie. It is a simple modification to the Backbone sync operations that allows it to communicate with a standard tastypie REST-API.

3.6.3 Satellite

Description

This module describes the architecture of how the backend will manage features related to managing Satellites.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.6.4 Devices

Description

This module describes the architecture of how the backend will manage features related to device management.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.6.5 Power Bill Guestimator

Description

This module describes the architecture of how the backend will manage features related to the power bill guestimator.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.6.6 Factory Reset

Description

This module describes the architecture of how the backend will manage features related to the factory reset.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.7 The Frontend

3.7.1 Authentication

Description

This module describes the architecture of how the frontend will create pages that allow the user to manage user and groups.

The authentication front end will consist of both the "login" page, and the user management pages. These pages either post directly to the server using standard form-submission methods (ie, not Ajax) or by using AJAX (if we have time to recreate the entire management system).

Program Flow

The user will first be presented with a login screen, which will take two pieces of information:

- Username - The user's username
- Password - The user's password

After these two pieces of information have been verified (ie, that password belongs to that user), the user will be logged in.

From here, the user has all kinds of options. The only ones that are interesting to this module is the settings panel, which includes a link to the user administration panel. The user administration panel will, initially, be a themed Django-admin view. Time permitting, this may be recreated to be a AJAX type interface.

Data Flow

All data will be sent to the server through POST requests, not through the REST API. This will allow us to skip over this module almost entirely. This is

not consistent with everything else, and will need to be remade later on if there is time.

Potential Problems

Themeing the admin interface to match our site will be tricky. At best, it will involve telling Django to include some extra stylesheets or using a different template. At worst, it will involve modifying the Django stylesheets directly. Either option works, but due to version control it would greatly preferred to be able to tell Django what styles to use.

3.7.2 Satellite

Description

This module describes the architecture of how the frontend will create pages that allow the user to add, remove, and modify Satellites associated with the system.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.7.3 Devices

Description

This module describes the architecture of how the frontend will create pages that allow the user to manage devices in the system.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.7.4 Frontend - View Data

Description

This module describes the architecture of how the frontend will create pages that allow the user to view various pieces of data in various formats.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.7.5 Power Bill Guestimator

Description

This module describes the architecture of how the frontend will create pages that allow the user to view guestimates of their monthly power bill, in various scenarios.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1

3.7.6 Factory Reset

Description

This module describes the architecture of how the frontend will create pages that allow the user to reset the system back to factory default settings.

Program Flow

Data Flow

UML Diagrams

Potential Problems

Sub-modules 1