SEADS Documentation

Release 1.0

Bryan Smith

June 02, 2015

CONTENTS

Author Bryan Smith

Note: Official API documentation is available here.

CHAPTER

ONE

INSTALLATION

1.1 Introduction

The SEADS web infrastructure provieds a simple API to read/write the data gathered by the SEAD Light.In addition, this framework is designed to provide in-house visualization and cost analysis of the energy usage. Built on Django, it is encouraged that this environment be augmented with new and improved applications that can interface with the data.

The current setup has all the necessary infrastructure in place for SEAD Lights to send their data. In addition, a proof-of-concept application has been created to interface with the data in a meaningful way.

There are two ways to deploy this system:

1. Scalable framework based in the Amazon Cloud Computing Services (recommended)
2. Atomic install on a single machine (easy)

1.2 Installing Scalable Framework in Amazon Compute Cloud (Ubuntu)

Note: This project was developed entirely on a Ubuntu build (14.04.2) so these instructions will be tailored towards that build. However, this framework should install semi-peacefully on any OS that can run python/nginx/uwsgi.

These instructions will assume you know how to interface with the amazon AWS console to create new instances. Refer to the Getting Started Guide for more information.

1.2.1 Basic Server Outline

For the scalable framework to work correctly, there is a bare minimum of 3 servers that need to be always running. Each server has a unique purpose:

1. Influxdb Server - A medium/large instance that houses the data from the SEAD Light. Needs to be large to handle the calculations that go into the fanout queries of the database.
2. Webapp Stateful Server - A server that houses the Django database that holds state information about the web application, such as user credentials and model relations.
3. Webapp Stateless Server - A skeleton server that serves as the frontend for users connecting to the web interface. This server is a clone of an image used in the auto scaling group.

The servers are setup with the following hierarchy:

The infrastructure is set up in this way to deal with a scalable load in an intelligent way. If there is little to no traffic to the website/REST API, the servers will spin down to a minimal state. However, if load increases, the infrastructure is designed to automatically spin up new instances of the web application so that no single instance is overloaded.

1.2.2 InfluxDB Server Setup

To get started, create a medium/large instance for the InfluxDB database. The operating system is recommended to be Ubuntu, but this is not a requirement. This server will only be interfaced by the stateless web servers under the following circumstances: 1) A user requests device data via the web application, 2) A user/device interacts with the REST API to read/write device data.

The following ports should be opened for the InfluxDB instance:

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	0.0.0.0/0	Self explanatory
Custom TCP Rule	TCP	8083	0.0.0.0/0	Exposes the database web API for interactive use
Custom TCP Rule	TCP	8086	0.0.0.0/0	Exposes the database REST port for the python interface

Once the server has booted, connect to it via ssh and do the following:

1. Install Git:

```
sudo apt-get install git
```

2. Clone the SEADS repository:

```
git clone https://github.com/Fraubluher/ShR2/
```

3. Run the deploy script for influxdb:

```
cd ShR2/Web\ Stack/
sudo ./deploy_database.sh
```

4. Reboot the server:

```
sudo reboot
```

5. Configure the database:

```
curl -X POST 'http://localhost:8086/db?u=<username>&p=<password>' \
          -d '{"name": "seads"}'
```

The InfluxDB server is now ready to respond to requests.

1.2.3 Django Web Application Stateful Server Setup

This process will walk you through the process of installing a stateful server for the Django web application.

Create a tiny/small instance running Ubuntu (the operating system is recommended to be Ubuntu, but this is not a requirement).

The following ports should be opened for the stateful server:

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	0.0.0.0/0	Self explanatory
MYSQL	TCP	3306	0.0.0.0/0	Port for remotely interfacing with Django database

Once the server has booted, connect to it via ssh and do the following:

1. Install Git:

```
sudo apt-get install git
```

2. Clone the SEADS repository:

```
git clone https://github.com/Fraubluher/ShR2/
```

3. Run the deploy script for influxdb:

```
cd ShR2/Web\ Stack/
sudo ./deploy_webapp_stateful.sh
```

This script will take you through the process of creating the MySQL database to be used by the stateless servers in the future. You will be prompted to create a root user on the database, remember the credentials for later.

This script will install all the necessary dependencies for the Django project. This will take a while, grab a beverage.

Near the end, several prompts will appear. You will be prompted to create the Django user in the MySQL database that is used to interface with the stateless servers. Leaving prompts blank will roll over to their default values indicated in the parentheses.

4. Reboot the server:

```
sudo reboot
```

This server should now be properly configured to run as a stateful implementation of the web application.

1.2.4 Django Web Application Stateless Server Setup

The final step in assembling the server infrastructure is to create a stateless instance of the web application. This will provide the basis for which an auto scaler can instantiate more/less instances of the web application automatically.

Create a tiny/small instance running Ubuntu (the operating system is recommended to be Ubuntu, but this is not a requirement).

The following ports should be opened for the stateful server:

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	0.0.0.0/0	Self explanatory
HTTP	TCP	80	0.0.0.0/0	Self explanatory

Since this is the forward-facing instance, the HTTP port is opened for clients to connect to. This allows both end users and SEAD Lights to connect and interact.

Once the server has booted, connect to it via ssh and do the following:

1. Install Git:

```
sudo apt-get install git
```

2. Clone the SEADS repository:

```
git clone https://github.com/Fraubluher/ShR2/
```

3. Run the deploy script for influxdb:

```
cd ShR2/Web\ Stack/
sudo ./deploy_webapp_stateless.sh
```

When this script runs, it will prompt for the address for the remote database (Django database host address). This is the address of the server created in the previous step.

4. Reboot the server:

```
sudo reboot
```

When the server reboots, you should now be able to connect to it from a web browser and test out the functionality. The stateless server is the address in which clients and SEAD Lights should connect.

1.2.5 Finishing Up

At this point, you have a functioning server framework that is eligible for load balancing and auto scaling. This guide does not get into the specifics since it is unique to the cloud service being used.

In general, these are the steps you should follow:

1. Create an image from the fully-configured webapp stateful server.
2. Configure and auto scaling group based on the image.
3. Configure a load balancer based off the auto scaling group.

If you choose to link the server's address to a domain name after configuring a load balancer, a CNAME record must be created with the DNS provider with the load balancer's address.

1.3 Installing Atomic Server

Note: This project was developed entirely on a Ubuntu build (14.04.2) so these instructions will be tailored towards that build. However, this framework should install semi-peacefully on any OS that can run python/nginx/uwsgi.

These instructions will not focus on deploying in the Amazon Compute Cloud, however it is certainly possible to do so.

1.3.1 Basic Server Outline

This server will comprise all aspects of the project on a single machine. This type of setup is intended for a small user base on the order of 10's of users. Any more and you should consider adopting the scalable approach above. It is recommended to use Ubuntu simply because this platform was developed and tested solely on Ubuntu.

To get started, open up the following ports on your machine:

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	0.0.0.0/0	Self explanatory
Custom TCP Rule	TCP	8083	0.0.0.0/0	Exposes the database web API for interactive use
Custom TCP Rule	TCP	8086	0.0.0.0/0	Exposes the database REST port for the python interface
MYSQL	TCP	3306	0.0.0.0/0	Port for remotely interfacing with Django database
HTTP	TCP	80	0.0.0.0/0	Self explanatory

1. Install Git:

```
sudo apt-get install git
```

2. Clone the SEADS repository:

```
git clone https://github.com/Fraubluher/ShR2/
```

3. Run the deploy script for influxdb:

```
cd ShR2/Web\ Stack/
sudo ./deploy_webapp_stateful.sh
```

This script will walk you through creating and configuring the databases needed. For any prompt asking for an address, enter 'localhost'.

4. Reboot the server:

```
sudo reboot
```

When the server reboots, verify it works by visiting the server from a webpage. All basic functionality should now exist.

CHAPTER

TWO

GETTING STARTED

Note: The SEAD Light will henceforth be referred to as a "Device" for sake of generalizing. This framework was built with the SEAD Light in mind, however any type of "device" is compatible if they follow the API Guidelines.

At this point, the project is now ready to begin accepting clients and devices.

To get devices connected to your project, simply point their database address at the server that faces the internet (for the scalable framework, this is the stateless server or load balancer). This will cause the devices to attempt to find themselves in the server, fail, then register themselves in the system.

From here, users can register to the website and pair to devices via their serial number.

2.1 API Documentation

You can interact with the server's API by navigating to /docs. This interface was designed to allow developers easy access to the framework so that they can learn it rapidly and integrate new types of devices into the database.

The REST API is set up to be very trusting. There are no checks for malicious behavior and we assume all users are benign. It is possible to alter any and all properties of a device such as changing the owner via the API. In the future, it is recommended to include API token authentication to prevent malicious attacks.

2.2 Administrative Interface

In addition to the front facing web interface, there is an administrative interface for managing database models directly. This interface was designed to allow an administrator the ability to alter the way the website functions without having to interface with the source code directly.

An administrator can use this interface to create/modify devices, circuit types, and appliance directly. This would presumably become useful when the algorithms on the SEAD Light mature to the point where disaggregation by appliance is feasable.

In addition, an administrator can also interface with the facets of the web application, including how event notifications are handled as well as add/modify interval notifications. These are the emails sent to users after a certain event has been detected or an interval has elapsed.

There is the ability to add/modify rate plans, territories, and utility companies to the web application, giving more realistic cost predictions for a user's device.

2.3 Connecting Devices to the Framework

With the new framework in place, we are now ready to begin connecting devices to the database. The general outline on how this is done is as follows:

1. Point device at the API Endpoint for your framework.

The endpoint of your system depends on whether or not the scalable framework is in place. If it is, then the endpoint is the address of the Load Balancer. If this is an atomic installation, then the endpoint is the address of the machine running the applications.

For the API already in place, this would be:

```
http://seads.io/api/
```

2. Query the devices database to check if the serial of the device is registered:

```
    GET http://seads.io/api/device-api/{serial}/
```

If the response is 404 (not found), continue to the next step. If the response is 302 (found), then the device is already connected and ready to transmit.

3. Register the device with the framework:

```
    POST http://seads.io/api/device-api/?serial={serial}
```

This POST request will register the device within the server, allocating database series and instantiating web application models that will relate to this device.

At this point, the device is ready to start transmitting packets. The device can transmit packets to the server without an owner. It is assumed that the owner is interested in seeing all data even before the device is paired, so the data is stored regardless of if the device is an orphan or not.

4. Begin data transmission:

```
    POST http://seads.io/api/device-api/
            {
                    "device": "/api/device-api/{serial}/",
                    "time"  : ["0x14d95894815", "0x1"],
                    "dataPoints":[
                                                {"wattage": 170},
                                                {"wattage": 169},
                                                ...
                    ]
            }
```

This is a typical packet that could be sent to the server from the device. Here is a breakdown of the fields:

- Device: The hyperlinked device sending the data.
- Time: A tuple of the format *[start_time, period]* in hexadecimal describing the packet's timing in milliseconds.
- dataPoints: An array of undefined size containing measured values for the server. The data points are numbered *0 ... n ... j* where nth data point has a timestamp of *start_time + n * period* and *j* is undefined.

CHAPTER

THREE

APPLICATIONS

Four distinct applications were created that encompass the SEAD web framework:

1. Microdata - Interfaces with the devices
2. Webapp - Interfaces with the clients
3. Farmer - Manages the devices
4. Home - Landing zone for new users

And (optionally) a fifth:

5. Debug - Models that assist in debugging the project

The applications were designed such that the Webapp application is hot-swappable. Since it is a proof of concept, it can either be added to or replaced altogether. Microdata and Farmer are able to run without Webapp, however their data cannot be interfaced with in a meaningful way without a web application.

Full Size

What follows is the breakdown of each application and how they work.

3.1 Microdata Application

This application is the direct interface between devices and the InfluxDB database. It is responsible for exposing the REST API endpoints.

A `microdata.models.Device` is what links to a SEAD Light out in the world. When a new SEAD connects to the system, it will check to see if it has been registered on the system by querying */api/device-api/{serial}/*. If the response is 404, the device will then register on the system as new with no owner. It is then the user's responsibility to pair the device via the web application in order to access the data.

Refer to the *Getting Started* guide for more information on interfacing the devices with the server.

3.1.1 Subpackages

microdata.management package

Subpackages

microdata.management.commands package

Submodules

microdata.management.commands.archive_database module

class microdata.management.commands.archive_database.**Command**

Bases: django.core.management.base.BaseCommand

args = ‘’

handle(**args*, ***options*)

help = ‘Backs up the data for each device relative to its retention policy.’

microdata.management.commands.archive_database.**safe_list_get**(*l*, *idx*, *default*)

microdata.management.commands.check_glacier_jobs module

class microdata.management.commands.check_glacier_jobs.**Command**

Bases: django.core.management.base.BaseCommand

handle(**args*, ***options*)

Module contents

Module contents

3.1.2 Submodules

3.1.3 microdata.admin module

Models registered to the administrative interface are listed below. These are the interfaces provided to an administrator that may have control over the system for a group of users.

class microdata.admin.**ApplianceAdmin**(*model*, *admin_site*)

Bases: django.contrib.admin.options.ModelAdmin

Class that allows administrator access to the Appliance models.

This class was included to give the administartor the ability to add new appliances as the algorithms detect them. This is exposed so that an administrator can extend the functionality of the Appliances without having to touch the source code.

list_display = (‘name’, ‘pk’, ‘serial’, ‘chart_color’)

media

class microdata.admin.**CircuitAdmin**(*model*, *admin_site*)

Bases: django.contrib.admin.options.ModelAdmin

list_display = (‘name’, ‘circuittype’, ‘pk’)

media

class microdata.admin.**CircuitTypeAdmin**(*model*, *admin_site*)

Bases: django.contrib.admin.options.ModelAdmin

list_display = (‘name’, ‘pk’)

media

class microdata.admin.**DeviceAdmin**(*model*, *admin_site*)

Bases: django.contrib.admin.options.ModelAdmin

This class gives an administrator direct access to the device model.

Most of the fields of the device model can be modified through the custom settings interface on the web application, but this interface gives the administrator direct control.

inlines = (<class 'microdata.admin.DeviceWebSettingsInline'>, <class 'farmer.admin.DeviceSettingsInline'>)

list_display = ('name', 'owner', 'serial', 'position', 'secret_key', 'registered', 'fanout_query_registered')

media

readonly_fields = ('secret_key',)

search_fields = ('name', 'serial')

class microdata.admin.**DeviceWebSettingsInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.StackedInline

can_delete = False

media

model
alias of DeviceWebSettings

verbose_name_plural = 'devicesettings'

3.1.4 microdata.models module

class microdata.models.**Appliance**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Describes a single Appliance. In the future, this model will have describing attributes that give the user helpful information when visualizing.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Appliance.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Appliance.**chart_color = None**
This field defines what color the chart will assign the appliance when it is being displayed. Modifiable via the admin interface.

Appliance.**circuittype_set**

Appliance.**eventnotification_set**

Appliance.**objects = <django.db.models.manager.Manager object>**

class microdata.models.**Circuit**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Most likely deprecated.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Circuit.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Circuit.**circuittype**

Circuit.**objects = <django.db.models.manager.Manager object>**

class microdata.models.**CircuitType**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Describes a Circuit Type. A Circuit Type is related to a list of *microdata.models.Appliance* and acts as a set of objects to discover within a circuit.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception CircuitType.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

CircuitType.**appliances**

CircuitType.**chart_color = None**
This field defines what color the chart will assign the circuit when it is being displayed. Modifiable via the admin interface.

CircuitType.**circuit_set**

CircuitType.**objects = <django.db.models.manager.Manager object>**

class microdata.models.**Device**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Describes a single Device owned by settings.AUTH_USER_MODEL.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Device.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Device.**channel_1**
The first of three channels of the SEAD Light. This is a design flaw and should be instead a ManyToManyField.

Device.**channel_2**
The second of three channels of the SEAD Light. This is a design flaw and should be instead a ManyToManyField.

Device.**channel_3**
The third of three channels of the SEAD Light. This is a design flaw and should be instead a ManyToManyField.

Device.**cost_daily = None**
The total cost calculated by the server today. Much cheaper to keep this in the django database than the InfluxDB.

Device.**data_retention_policy = None**
The amount of time the data from this device can live in the database. Anything older will be archived to Amazon Glacier.

Device.**delete**(**args*, ***kwargs*)
Custom delete method.

Drop the series from the influxdb database.

Device.**devicesettings**

Device.**devicewebsettings**

Device.**event_set**

Device.**fanout_query_registered** = None
A true/false that is set when a device is created indicating the continuous queries in the database have been registered.

Device.**ip_address** = None
This field is no longer actively used. This was a proof of concept for early device communication. Deprecated.

Device.**kilowatt_hours_daily** = None
A counter that is reset by a cron job once a day that keeps an accumulation of the total kwh this device has measured over the course of a day.

Device.**kilowatt_hours_monthly** = None
A counter that is reset by a cron job once a month that keeps an accumulation of the total kwh this device has measured over the course of a month.

Device.**name** = None
A non-unique name field for a device. This field is solely for user experience, it has no functional purpose.

Device.**objects** = <django.db.models.manager.Manager object>

Device.**owner**
A Foreign key relation. We use this relation to pair a device to a user on the web application.

Device.**registered** = None
Synonym for paired. This protects againts users trying to pair an already paired device.

Device.**save**(***kwargs*)
Custom save method.

This function will do several things if it has not done so already:

- Create a secret key. 3 digits followed by 4 letters. Not currently in use (could be used for pairing devices)
- Register fanout queries in database.
- Device name cannot be None, default is "Device <serial>".
- Create a *farmer.models.DeviceSettings* object.
- Create a *webapp.models.DeviceWebSettings* object.
- Give default values to the channels.

Device.**secret_key** = None
This field is not currently in use, but the functionality exists. This field was intended to be used to pair a device to a user.

Device.**serial** = None
The primary key of the model. This is what the device uses to interface to the API and how users currently pair a device.

Device.**share_with**
This field acts much like an owner, however share_with users cannot alter device settings.

class microdata.models.**Event**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Generic class to catch the Event REST Packets from devices. Relates to a *microdata.models.Device*.

These models are not stored on the Django database since they are converted to InfluxDB.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Event.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Event.**dataPoints = None**
An array of undefined size containing measured values for the server. The data points are numbered *0 ... n ... j* where nth data point has a timestamp of *start_time + n * period* and *j* is undefined.

Event.**device**
This is the relation between the device and its data. This is established when the device specifies its hyperlinked model via the REST call.

Event.**frequency = None**
The frequency, in Hertz, of the packet's data points. Used to calculate the offset of all points.

Event.**objects = <django.db.models.manager.Manager object>**

Event.**query = None**
Deprecated. This field is useful for debugging the REST requests, but since the model is not saved, this is a volatile field.

Event.**save**(***kwargs*)
Custom save method.

This method is the powerhouse of the API. It can take an array of data points from a device and convert them into database entries in InfluxDB.

The method will also keep a running count of how many kwh have been consumed this day and this month. If it exceeds the allotted kwh for the device's tier, advance the tier a level.

If the data coming in is sufficiently in the past such that the database will not calculate its mean value, refresh the query to trigger a backfill of the data.

When a model is being saved, it has already been created by *microdata.views.EventViewSet*.

The Event is parsed as follows:

```
start = self.start
frequency = self.frequency
count = 0

for point in dataPoints:
   time = start + count * (1/frequency)
   db.write_points(time, wattage)
```

Event.**start = None**
The start time, in milliseconds, of the first data point in the packet. Used to calculate offset of all proceeding points.

3.1.5 microdata.serializers module

class microdata.serializers.**ApplianceSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

fields = ('name', 'serial', 'chart_color')

model
alias of Appliance

class microdata.serializers.**CircuitSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

fields = ('name', 'appliances', 'chart_color', 'pk')

model
alias of CircuitType

class microdata.serializers.**DeviceSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

fields = ('owner', 'ip_address', 'secret_key', 'serial', 'name', 'registered', 'fanout_query_registered', 'channel_1',

model
alias of Device

class microdata.serializers.**EventSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

fields = ('device', 'dataPoints')

model
alias of Event

3.1.6 microdata.tests module

3.1.7 microdata.views module

class microdata.views.**ApplianceViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

API endpoint that allows appliances to be viewed or edited.

queryset

serializer_class
alias of ApplianceSerializer

class microdata.views.**CircuitViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

API endpoint that allows circuits to be viewed or edited.

queryset

serializer_class
alias of CircuitSerializer

class microdata.views.**DeviceViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

API endpoint that allows devices to be viewed or edited.

create(*request*)

queryset

serializer_class
alias of DeviceSerializer

class microdata.views.**EventViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

API endpoint that allows events to be viewed or edited.

create(*request*)
Custom create method.

Used to parse the packets sent via the REST API. Since the packets are in a JSON array, the Django REST Framework has no native way of handling these, so we do it ourselves.

queryset

serializer_class
alias of EventSerializer

class microdata.views.**KeyForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*)
Bases: django.forms.forms.Form

Form class used to generate a simple key form. Currently used when a user intends to add a new device via the webapp interface.

base_fields = {'serial': <django.forms.fields.IntegerField object at 0x2ba7e1df4bd0>}

media

microdata.views.**initiate_job_to_glacier**(*request*, *requester*, *end_time*)
Experimental class to demonstrate the possibility to archive old data to Amazon Glacier.

Requires an Amazon AWS key to be set in the environment variables.

microdata.views.**new_device**(*request*)
Function used to service a user's request to add a new *microdata.models.Device*.

Context

form *microdata.views.KeyForm* object if user requests the form

error string - error description if present

created true/false if device is created

Device serialized *microdata.models.Device* object if *microdata.models.Device* is created

Templates:

base/new_device/key.html

base/new_device/help.html

base/new_device/first.html

base/new_device/result.html

microdata.views.**timestamp**(*request*)
Function to return the server's time in milliseconds.

This function is possibly deprecated. Devices should now get the server time from farmer.DeviceSettingsViewSet.

3.1.8 Module contents

3.2 Webapp Application

This application is what interfaces with the data that is retrieved by the Microdata application. Developed as a sort of proof-of-concept, this application has the minimal functionality required to visualize the data coming from the devices in a somewhat meaningful way.

This application is responsible for serving requests from clients on the web by interfacing with the database to display the data. Many of the facets of this application were designed to be interacted with via AJAX, giving the dashboard the responsiveness necessary to provide a useful user experience. Some of the calls to the database can be rather costly in terms of time, so they are lazy loaded, as in only when needed.

In addition, the webapp module houses the HTML and Javascript needed to run the web interface on the client side.

3.2.1 Subpackages

webapp.management package

Subpackages

webapp.management.commands package

Submodules

webapp.management.commands.email_event module

webapp.management.commands.email_interval module

webapp.management.commands.reset_kilowatt_accumulations module
class webapp.management.commands.reset_kilowatt_accumulations.**Command**
Bases: django.core.management.base.BaseCommand

args = 'daily, weekly'

handle(**args*, ***options*)

help = ''

Module contents

Module contents

3.2.2 Submodules

3.2.3 webapp.admin module

class webapp.admin.**DashboardSettingsInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.StackedInline

can_delete = False

media

model
alias of DashboardSettings

verbose_name_plural = 'dashboardsettings'

class webapp.admin.**RatePlanAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

Class used to expose the *webapp.models.RatePlan* model to the administrator. This was designed to be flexible to allow an administrator to add/modify Rate Plans based on electric company data.

inlines = (<class 'webapp.admin.TierInline'>,)

list_display = ('description', 'utility_company', 'pk')

media

class webapp.admin.**TerritoryAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

Class used to expose the webapp.models.Territory ` model to the administrator. This was designed to be flexible to allow an administrator to add/modify Territories based on electric company data.

list_display = ('description', 'rate_plan', 'pk')

media

class webapp.admin.**TierInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.StackedInline

Class used to expose the *webapp.models.Tier* model to the administrator. This was designed to be flexible to allow an administrator to add/modify tiers based on electric company data.

media

model
alias of Tier

class webapp.admin.**UserAdmin**(*model*, *admin_site*)
Bases: django.contrib.auth.admin.UserAdmin

inlines = (<class 'webapp.admin.UserSettingsInline'>, <class 'webapp.admin.DashboardSettingsInline'>)

media

class webapp.admin.**UserSettingsInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.StackedInline

can_delete = False

media

model
alias of UserSettings

verbose_name_plural = 'usersettings'

class webapp.admin.**UtilityCompanyAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

list_display = ('description', 'pk')

media

3.2.4 webapp.device_dictionary module

3.2.5 webapp.models module

class webapp.models.**DashboardSettings**(*id*, *user_id*, *stack*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception DashboardSettings.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

DashboardSettings.**objects = <django.db.models.manager.Manager object>**

DashboardSettings.**user**

class webapp.models.**DeviceWebSettings**(*id*, *device_id*, *current_tier_id*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception DeviceWebSettings.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

DeviceWebSettings.**current_tier**

DeviceWebSettings.**device**

DeviceWebSettings.**objects = <django.db.models.manager.Manager object>**

DeviceWebSettings.**rate_plans**

DeviceWebSettings.**territories**

DeviceWebSettings.**utility_companies**

class webapp.models.**EventNotification**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Notification sent to users via email whenever a notable event is detected.

This class is not currently in use since the system is not set up in such a way as to detect any events. However, the notification framework is in place such that when the functionality is added, this class should be called in response to an event.

These notifications can be added/modified via the admin interface.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception EventNotification.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

EventNotification.**appliances_to_watch**
Assemble a group of appliances to watch. Could be one or many.

EventNotification.**description = None**
The description of the event notification as a user would see it when selecting/deselecting the notification in the settings interface

EventNotification.**email_subject = None**
An email-friendly subject for the event notification.

EventNotification.**keyword = None**
Used to trigger the event notification in the django manager.

EventNotification.**objects = <django.db.models.manager.Manager object>**

EventNotification.**period_of_time = None**
Proof of concept field to provide a threshold. If a group of appliances surpasses the threshold for a period of time, then send the email.

EventNotification.**usersettings_set**

EventNotification.**watts_above_average = None**
Proof of concept field to provide a threshold. If a group of appliances surpasses the threshold for a period of time, then send the email.

class webapp.models.**IntervalNotification**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception IntervalNotification.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

IntervalNotification.**notification_set**

IntervalNotification.**objects = <django.db.models.manager.Manager object>**

IntervalNotification.**usersettings_set**

class webapp.models.**Notification**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

DEPRECATED

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Notification.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Notification.**interval_notification**

Notification.**objects = <django.db.models.manager.Manager object>**

Notification.**user**

class webapp.models.**RatePlan**(*id*, *utility_company_id*, *description*, *data_source*, *min_charge_rate*, *california_climate_credit*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception RatePlan.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

RatePlan.**devicewebsettings_set**

RatePlan.**objects = <django.db.models.manager.Manager object>**

RatePlan.**territory_set**

RatePlan.**tier_set**

RatePlan.**utility_company**

class webapp.models.**Territory**(*id*, *rate_plan_id*, *description*, *data_source*, *summer_start*, *winter_start*, *summer_rate*, *winter_rate*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Territory.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Territory.**devicewebsettings_set**

Territory.**objects = <django.db.models.manager.Manager object>**

Territory.**rate_plan**

class webapp.models.**Tier**(*id*, *rate_plan_id*, *tier_level*, *max_percentage_of_baseline*, *rate*, *chart_color*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Tier.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Tier.**devicewebsettings_set**

Tier.**objects = <django.db.models.manager.Manager object>**

Tier.**rate_plan**

class webapp.models.**UserSettings**(*id*, *user_id*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception UserSettings.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

UserSettings.**event_notification**

UserSettings.**interval_notification**

UserSettings.**objects = <django.db.models.manager.Manager object>**

UserSettings.**user**

class webapp.models.**UtilityCompany**(*id*, *description*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception UtilityCompany.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

UtilityCompany.**devicewebsettings_set**

UtilityCompany.**objects = <django.db.models.manager.Manager object>**

UtilityCompany.**rateplan_set**

3.2.6 webapp.tests module

3.2.7 webapp.timeseries module

webapp.timeseries.**smooth**(*x*, *window_len=11*, *window='hanning'*)
smooth the data using a window with requested size.

This method is based on the convolution of a scaled window with the signal. The signal is prepared by introducing reflected copies of the signal (with the window size) in both ends so that transient parts are minimized in the begining and end part of the output signal.

input: x: the input signal window_len: the dimension of the smoothing window; should be an odd integer window: the type of window from 'flat', 'hanning', 'hamming', 'bartlett', 'blackman'

flat window will produce a moving average smoothing.

output: the smoothed signal

example:

t=linspace(-2,2,0.1) x=sin(t)+randn(len(t))*0.1 y=smooth(x)

see also:

numpy.hanning, numpy.hamming, numpy.bartlett, numpy.blackman, numpy.convolve scipy.signal.lfilter

TODO: the window parameter could be the window itself if an array instead of a string NOTE: length(output) != length(input), to correct this: return y[(window_len/2-1):-(window_len/2)] instead of just y.

http://wiki.scipy.org/Cookbook/SignalSmooth

3.2.8 webapp.views module

class webapp.views.**Object**(*serial*)

class webapp.views.**SettingsForm**(**args*, ***kwargs*)
Bases: django.forms.forms.Form

base_fields = {'new_username': <django.forms.fields.CharField object at 0x2ba7e26077d0>, 'password1': <django.fo

channel_1_choices = []

channel_2_choices = []

channel_3_choices = []

clean_password2()

error_messages = {'password_mismatch': "The two password fields didn't match."}

get_notifications(**args*, ***kwargs*)

get_rate_plans(**args*, ***kwargs*)

get_territories(**args*, ***kwargs*)

get_utility_companies(**args*, ***kwargs*)

media

notification_choices = []

rate_plan_choices = []

share_with_choices = []

territory_choices = []

utility_company_choices = []

webapp.views.**billing_information**(*request*, **args*, ***kwargs*)

webapp.views.**chartify**(*data*)

webapp.views.**charts_deprecated**(*request*, **args*, ***kwargs*)

webapp.views.**circuits_information**(*request*, **args*, ***kwargs*)

webapp.views.**dashboard**(*request*, **args*, ***kwargs*)

webapp.views.**dashboard_update**(*request*, **args*, ***kwargs*)

webapp.views.**default_chart**(*request*, **args*, ***kwargs*)

webapp.views.**device_chart**(*request*, **args*, ***kwargs*)

webapp.views.**device_data**(*request*, **args*, ***kwargs*)

webapp.views.**device_is_online**(*device*)

webapp.views.**device_location**(*request*, **args*, ***kwargs*)

webapp.views.**device_status**(*request*, **args*, ***kwargs*)

webapp.views.**export_data**(*request*, **args*, ***kwargs*)

webapp.views.**generate_average_wattage_usage**(*request*, **args*, ***kwargs*)

webapp.views.**generate_heatmap_data**(*serial*)

webapp.views.**get_wattage_usage**(*request*, **args*, ***kwargs*)

webapp.views.**group_by_mean**(*serial*, *unit*, *start*, *stop*, *localtime*, *circuit_pk*)

webapp.views.**heatmap**(*request*, **args*, ***kwargs*)

webapp.views.**landing**(*request*, **args*, ***kwargs*)

webapp.views.**make_choices**(*querysets*)

webapp.views.**merge_subs**(*lst_of_lsts*)

webapp.views.**remove_device**(*request*, **args*, ***kwargs*)

webapp.views.**settings**(**args*, ***kwargs*)

webapp.views.**settings_account**(*request*, **args*, ***kwargs*)

webapp.views.**settings_change_device**(*request*, **args*, ***kwargs*)

webapp.views.**settings_dashboard**(*request*, **args*, ***kwargs*)

webapp.views.**settings_device**(*request*, **args*, ***kwargs*)

3.2.9 Module contents

3.3 farmer package

3.3.1 Submodules

3.3.2 farmer.admin module

class farmer.admin.**DeviceSettingsAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

media

verbose_name_plural = 'devicesettings'

class farmer.admin.**DeviceSettingsInline**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.StackedInline

can_delete = False

media

model
alias of DeviceSettings

3.3.3 farmer.models module

class farmer.models.**DeviceSettings**(*device_id*, *device_serial*, *main_channel*, *adc_sample_rate*, *transmission_rate_milliseconds*, *date_now*)
Bases: django.db.models.base.Model

CHANNEL_CHOICES = ((1, 'Channel 1'), (2, 'Channel 2'), (3, 'Channel 3'), (4, 'Channel 4'))

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception DeviceSettings.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

DeviceSettings.**SAMPLE_RATE_CHOICES = ((0, 125000), (1, 62500), (2, 31250), (3, 15625), (4, 7812.5), (5, 3906.25), (**

DeviceSettings.**device**

DeviceSettings.**get_adc_sample_rate_display**(**moreargs*, ***morekwargs*)

DeviceSettings.**get_main_channel_display**(**moreargs*, ***morekwargs*)

DeviceSettings.**objects = <django.db.models.manager.Manager object>**

DeviceSettings.**save**(***kwargs*)

3.3.4 farmer.serializers module

class farmer.serializers.**DeviceSettingsSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.ModelSerializer

class Meta

fields = ('device', 'device_serial', 'main_channel', 'transmission_rate_milliseconds', 'adc_sample_rate', 'date_nov

model
alias of DeviceSettings

3.3.5 farmer.tests module

3.3.6 farmer.views module

class farmer.views.**DeviceSettingsViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

API endpoint that allows devicesettings to be viewed or edited.

list(*request*)

queryset

retrieve(*request*, *pk=None*)

serializer_class
alias of DeviceSettingsSerializer

3.3.7 Module contents

3.4 home package

3.4.1 Subpackages

3.4.2 Submodules

3.4.3 home.admin module

3.4.4 home.models module

3.4.5 home.serializers module

class home.serializers.**UserSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: rest_framework.serializers.HyperlinkedModelSerializer

class Meta

fields = ('url', 'username', 'email', 'is_staff')

model
alias of User

3.4.6 home.tests module

3.4.7 home.views module

class home.views.**UserViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ModelViewSet

queryset

serializer_class
alias of UserSerializer

home.views.**account**(*request*)

home.views.**index**(*request*)

home.views.**register**(*request*)

home.views.**signin**(*request*)

home.views.**signout**(*request*)

3.4.8 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

f

h

m

w

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W