RPyMostat Documentation

Release 0.1.0

Jason Antman

May 28, 2017

Contents

Master: Develop: RPyMostat - A python-based modular intelligent home thermostat, targeted at (but not requiring) the RaspberryPi and similar small computers, with a documented API.

CHAPTER 1

Contents

Planning

A python-based intelligent home thermostat, targeted at (but not requiring) the RaspberryPi and similar small computers. (Originally "RaspberryPyMostat", for 'RaspberryPi Python Thermostat', but that's too long to reasonably name a Python package).

Especially since the introduction of the Nest thermostat, a lot of people have attempted a project like this. I'd like to think that mine is different - perhaps more polished, perhaps it stores historical data in a real, logical way. Multiple temperatures are nice, and the pluggable scheduling and decision engines are something I haven't seen in any others yet. The completely open API, and the fact that some of the out-of-the-box components use it is new too. And after looking at some of the options out there, I think the idea of it being packaged and distributed properly is pretty novel too, as are my hopes for a platform-agnostic system; a lot of the options out there are really hardware-hacking projects, and I want to make software that works with as many hardware options as it can. But when it comes down to it, this is an idea that I tried a long time ago and never finished, and want to have another try at regardless of whether it does something unique or becomes just another one of the hundred pieces of software that do the same thing. I'm also going to be playing with some technology that I've never used before, so for me this is as much about learning and exploring as it is about producing a polished final codebase.

See:

- Architecture.md for an overview of the architecture, and most of the documentation that currently exists.
- DISCOVERY.md for some information on service discovery
- TWISTED.md for some docs on using Twisted for this

Features

Features planned for the initial release

- Flexible rules-based scheduling. This can include cron-like schedules (do X at a given time of day, or time of day on one or more days of week, etc.), one-time schedule overrides ("I'm going to be away from December 21st to 28th this year, just keep the temperature above Y"), or instant adjustments ("make the temperature X

degress NOW", in the web UI). The most specific schedule wins. Inital scheduling will support some mix of what can be represented by ISO8601 time intervals and cron expressions.

- Support for N temperature sensors, and scheduling based on them; i.e. set a daytime target temperature based on the temperature of your office, and a nighttime target based on the temperature in the bedroom.
- Web UI with robust mobile and touch support. Ideally, the entire system should be configurable by a web UI once it's installed (which should be done with a Puppet module).
- Some sort of physical on-the-wall touchscreen control, using the web UI.
- Everything AGPL 3.0.
- Scheduling and decision (system run) implemented in plugins (packages, entry points) that use a defined API; some way of reflecting this in the Web UI (maybe this should come over the master API). Initially just implement scheduling as described above and setting temperature based on one temp input; subsequent plugins could include averaging across multiple inputs, weighted average, and predictive on/off cycles (including outside temperature input).
- Support running all on one RPi, or splitting components apart; should support as many OSes as possible. Support for smaller devices as temperature sensors would be nice.
- Microservice/component architecture.
- Open, documented APIs. Aside from the main engine, it should be possible to implement the other components in other languages.
- mDNS / DNS-SD for zero configuration on devices other than the engine.

Features planned for future releases

- Data on current and desired temperature(s) and heating/cooling state will be collected. This should allow the scheduling engine to build up historical data on how long it takes to heat or cool one degree at a given temperature, and should allow us to trigger heating/cooling to reach the scheduled temperature at the scheduled time (as opposed to starting the heating/cooling at the scheduled time).
- Historical data stored in some time-series database; should include all temperature values at the beginning of a run, and every X minutes during a run.

Relevant Links / Similar Projects

- https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/temperature/
- https://www.adafruit.com/product/1012
- http://www.projects.privateeyepi.com/home/temperature-gauge
- http://m.instructables.com/id/Raspberry-Pi-Temperature-Humidity-Network-Monitor/
- Raspberry Pi Thermostat Part 1: System Overview - The Nooganeer
- Willseph/RaspberryPiThermostat
- python - Thermostat Control Algorithms - Stack Overflow
- VE2ZAZ - Smart Thermostat on the Raspberry Pi
- Raspberry Pi • View topic - Web enabled thermostat project
- Rubustat - the Raspberry Pi Thermostat | Wyatt Winters | Saving the world one computer at a time
- Makeatronics: Raspberry Pi Thermostat Hookups

- Makeatronics: Thermostat Software
- Willseph/RaspberryPiThermostat: A Raspberry Pi-powered smart thermostat written in Python and PHP. - Python sensors and control but PHP LAMP web UI. MIT license. Looks like it's got a good bit of information, especially on wiring/setup and photos of the install on Imgur.
- ianmtaylor1/thermostat: Raspberry Pi Thermostat code - Python project that reads 1-wire temps and uses SQLAlchemy. Relatively simple beyond that.
- chaeron/thermostat: Raspberry Pi Thermostat - Fairly nice touchscreen UI and pretty complete, but one untested python file and only one physical piece.
- mharizanov/ESP8266_Relay_Board: Three Channel WiFi Relay/Thermostat Board - firmware source code and hardware designs for a WiFi relay/thermostat board. Probably won't use this, but interesting.
- mdarty/thermostat: Raspberry Pi Thermostat Controller - python/flask app for a Python RPi thermostat.
- tom91136/thermostat: A simple thermostat for RaspberryPi written in Python - Another Flask, DS18B20 thermostat with GPIO relays.
- jeffmcfadden/PiThermostat: Build a Raspberry Pi Thermostat - Rails app for an RPi thermostat.
- Forever-Young/thermostat-web: Django application for thermostat control - single-host
- wywin/Rubustat: A thermostat controller for Raspberry Pi on Flask
- tommybobbins/PiThermostat: Raspberry Pi, TMP102 and 433 Transmitter to make an Redis based Central heating system - Redis-based system using Google Calendar for scheduling
- jpardobl/django-thermostat: Django app to control a heater
- tinkerjs/Pi-Thermostat: A Raspberry Pi based thermostat - Python and RPi, but single-host. Blog post has some nice diagrams, pictures, and information on HVAC systems.
- cakofony/thermostat: Web enabled thermostat project to run on the raspberry pi. - Python, includes support for an Adafruit character LCD display.
- Raspberry Pi Thermostat Part 1: System Overview - The Nooganeer - nice web UI demo
- VE2ZAZ - Smart Thermostat on the Raspberry Pi - Flask UI
- openHAB - JVM-based, vendor-agnostic home automation "hub". Includes web UI. Rule creation appears to be via a Java UI though.
- home-assistant/home-assistant: Open-source home automation platform running on Python 3 - Python3 home automation server with web UI. Looks like it could be really interesting, but not sure how much support it has for the advanced scheduling I want.
- WTherm – a smart thermostat | NiekProductions - Arduino, PHP but has some good concepts.
- Home | pimatic - smart home automation for the raspberry pi - node.js home automation framework. Once again, doesn't have support for the kind of scheduling I want.
- Matt Brenner / PyStat · GitLab - multi-threaded Ptrhon thermostat; Flask, RPi. screenshots. Looks nice, but doesn't seem to have the type of scheduling I want, and runs as a single process/single host.
- serial_device2 - Extends serial.Serial to add methods such as auto discovery of available serial ports in Linux, Windows, and Mac OS X
- pyusbg2 - PyUSB offers easy USB devices communication in Python. It should work without additional code in any environment with Python >= 2.4, ctypes and an pre-built usb backend library (currently, libusb 0.1.x, libusb 1.x, and OpenUSB).

Some Technical Bits and Questions

API

- raml - RESTful API Modeling Language
- architecting version-less APIs

Engine

- The main process will likely have to have a number of threads: API serving (ReST API), timer/cron for scheduling and comparing temp values to thresholds, main thread (am I missing anything?)
- Use workers (either real Celery, or just async calling a process/thread) to calculate things?
- schedules and overrides
- schedules have start and end time, that are cron-like
- overrides have a specific start time, and end time that's either specific (input can be a specific datetime, or a duration) or when the next schedule starts
- backend - when a schedule or override is input, backend recalculates the next X hours of instructions (schedule with overrides applied), caches them, makes them accessible via API
- schedules and overrides
- default temperature thresholds (how much over/under to trigger/overshoot and how often to run)
- schedules/overrides have temperature targets and thresholds - which sensors to look at, how to weight them. Can be a "simple" input (look at only one sensor, one target temp) or a weighted combination. Can save a default calculation method/sensor weighting.
- make sure we don't start/stop the system too often

UI

- Web UI will probably use Flask, **TODO:** but I need to figure out how easy it is to get that to just wrap an API.
- **TODO:** Is there any way that we can generate (dynamically? code generation?) the API server and client? The web UI? Is there an existing web UI "thing" to just wrap a ReST API? Would this help testing?
- I know some of the python API clients I've worked with do this... I just need to figure out how, because it's an area I've never really looked into.
- Just provide a pretty (or usable) wrapper around the decision engine API. Honestly I'd love it if this could be generated entirely dynamically - i.e. the decision engine's plugins know about some input data types, and the web UI knows how to render them. The web UI is just a pile of components, and pulls information about what it needs dynamically from the decision engine. That's really complicated to implement, but OTOH, I'm not sure how else we allow pluggable scheduling and decision modules.
- visual schedule overlay like PagerDuty
- [tastejs/todomvc: Helping you select an MV* framework - Todo apps for Backbone.js, Ember.js, AngularJS, and many more](https://github.com/tastejs/todomvc) / [TodoMVC](http://todomvc.com/)
- https://en.wikipedia.org/wiki/HATEOAS

- **looks good** - [Writing a Javascript REST client - miguelgrinberg.com](http://blog.miguelgrinberg.com/post/writing-a-javascript-rest-client) - [Twitter Bootstrap](http://twitter.github.io/bootstrap/) for presentation (see [fluid layout model](http://getbootstrap.com/2.3.2/examples/fluid.html)), [Knockout](http://knockoutjs.com/) for MVC.
- [vinta/awesome-python: A curated list of awesome Python frameworks, libraries, software and resources](https://github.com/vinta/awesome-python#database-drivers)
- [Ajenti Core - a Web-UI Toolkit](http://ajenti.org/core/) - has a really nice UI, and is Python on the backend
- [Backbone.js](http://backbonejs.org/) - might be good... it's an in-browser MVC. A little worried about memory use.
- [Creating a Single Page Todo App with Node and Angular | Scotch](https://scotch.io/tutorials/creating-a-single-page-todo-app-with-node-and-angular)

Testing

- Unit tests should mock out the txmongo connection. Integration tests require Mongo, and should run a Docker container of it. Need to look into how to do this nicely on Travis.
- We'll need some real data fixtures, and to look into the right way to dump and load data from/to Mongo.
- Assuming we're going with the API-based model, unit tests should be simple. Integration and acceptance tests are another question.
- **TODO:** How to test the API server and client?
- **TODO:** How to test the separate services, in isolation from the server?
- **TODO:** Try to find a strong unit testing framework for the web UI; we can deal with integration/acceptance testing later.
- **TODO:** How do I do acceptance/integration testing with service discovery if I have this running (like, in my house) on my LAN? Just use some "system number" variable?

Relay/Physical Control Unit

dead-simple:

1. Process starts up, uses service discovery to find the decision engine.
2. Registers itself with some sort of unique ID (hardware UUID, RaspberryPi serial number, etc.)
3. Discovers available relay outputs and their states, assigns a unique ID to each.
4. POST this information to the decision engine.
5. Start a web server.
6. Wait for an API request from the decision engine, which is either a GET (current status) or POST (set state).

Decision Engine / Master Control Process

Here's where the complexity lies.

- Keep (time-series?) database of historical data on temperature, system state, etc. (including data required for predictive system operation)
- Determine the current and next (N) schedules.

- Constantly (every N seconds) compare temperature data to current schedule and operate system accordingly
- Re-read schedules whenever a change takes place
- Show end-user current system state and upcoming schedules
- Provide a plugin interface for schedule algorithms
- Provide a plugin interface for decision (system run/stop) algorithms
- Support third-party web UIs via its API, which needs to include support for the plug-in scheduling and decision algorithms (which exist only in this process, not the web UI)
- Support versioning of ReST and internal APIs

Datastore

MongoDB 2.4. Raspbian has it for ARM.

- txmongo and its docs
- txmongo twisted.web example

Time-Series:

- Schema Design for Time Series Data in MongoD - MongoDB Blog
- Time Series
- MongoDB for Time Series Data Part 1: Setting the Stage for Sensor Management | MongoDB
- MongoDB for Time Series Data | MongoDB
- Efficient storage of non-periodic time series with MongoDB
- Capped Collections — MongoDB Manual 3.2
- MongoDB tech behind our time series graphs - 30TB per month
- Make Interactive Time Series Charts for IoT Using Live MongoDB Data | SlamData
- Storing time-series data with MongoDB and TokuMX
- MongoDB Time Series: Introducing the Aggregation Framework - DZone Database
- comSysto Blog: Processing and analysing sensor data
- MongoDB time series: Introducing the aggregation framework | Vlad Mihalcea's Blog

Physical Control Interface

- Wall mount tablet for the UI? There's some cheap ones, and AutoStart - No root - Android Apps on Google Play to autostart an app (browser) at boot...
- Wall mount touchscreens: - https://www.adafruit.com/products/1892 - https://www.adafruit.com/products/2033 - https://www.adafruit.com/products/2534 - https://www.adafruit.com/products/2260 - Could just use an old phone for now... or set it up somewhere on a bookcase or table... - https://blog.adafruit.com/2014/09/05/wall-mounted-touchscreen-raspberry-pi-home-server-piday-raspberrypi-raspberry_pi/ - http://www.neosecsolutions.com//products.php?62&cPath=21 - http://www.modmypi.com/blog/raspberry-pi-7-touch-sreen-display-case-assembly-instructions - http://www.thingiverse.com/thing:1082431 - http://www.thingiverse.com/thing:1034194 - https://www.element14.com/community/docs/DOC-78156/l/raspberry-pi-7-touchscreen-display

- Pi3 Model B - $35-40 - - https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ - wifi (2.4GHz 802.11n??? - might need USB?) - USB - GPIO - HDMI - DSI display interface
- Pi Zero - https://www.raspberrypi.org/products/pi-zero/ - sold out everywhere :(- Mini HDMI - USB On-The-Go - MicroUSB power - HAT-compatible 40-pin header - onboard wifi hack: https://www.raspberrypi.org/forums/viewtopic.php?f=63&t=127449 - starter kit - https://www.adafruit.com/products/2816 - would need USB WiFi dongle and GPIO sensors
- RPi DS18B20 - https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/temperature/ - https://learn.adafruit.com/adafruits-raspberry-pi-lesson-11-ds18b20-temperature-sensing/hardware - http://www.modmypi.com/blog/ds18b20-one-wire-digital-temperature-sensor-and-the-raspberry-pi - https://www.raspberrypi.org/forums/viewtopic.php?t=54238&p=431812

Other Hardware

- Miniature WiFi 802.11b/g/n Module: For Raspberry Pi and more ID: 814 - $11.95 : Adafruit Industries, Unique & fun DIY electronics and kits
- USB WiFi 802.11b/g/n Module: For Raspberry Pi and more ID: 1012 - $12.95 : Adafruit Industries, Unique & fun DIY electronics and kits
- Assembled Pi Cobbler Plus - Breakout Cable for Pi B+/A+/Pi 2/Pi 3 ID: 2029 - $6.95 : Adafruit Industries, Unique & fun DIY electronics and kits
- Assembled Pi T-Cobbler Plus - GPIO Breakout for RasPi A+/B+/Pi 2/Pi 3 ID: 2028 - $7.95 : Adafruit Industries, Unique & fun DIY electronics and kits
- GPIO Header for Raspberry Pi A+/B+/Pi 2/Pi 3 2x20 Female Header ID: 2222 - $1.50 : Adafruit Industries, Unique & fun DIY electronics and kits
- 0.1 2x20-pin Strip Right Angle Female Header ID: 2823 - $1.50 : Adafruit Industries, Unique & fun DIY electronics and kits

RPyMostat Reference Implementation

My planned reference implementation of the system is:

- RaspberryPi 2+ physical control unit - USB relay output for control, and a temperature sensor, connecting via WiFi.
  - DS18B20 temperature sensor using GPIO
  - For system control, either a PiFace or a Phidgets 1014 USB 4 relay kit, both of which I already have.
- 2x RaspberryPi Zero temperature sensors in other rooms, connecting via WiFi.
  - DS18B20 temperature sensor using GPIO
- Engine, web UI and a third (USB OWFS) temperature input on my desktop computer.
  - DS18S20 temperature sensor connected via DS9490R usb-to-1-wire adapter

RPyMostat Installation, Configuration and Usage

Requirements

- Python 2.7 or 3.3+ (currently tested with 2.7, 3.3, 3.4, 3.5; tested on pypy but does not support pypy3)

- Python VirtualEnv and `pip` (recommended installation method; your OS/distribution should have packages for these)
- MongoDB (developed against 2.4, which is available in the Debian and Raspbian repos)

Installation

It's recommended that you install into a virtual environment (virtualenv / venv). See the virtualenv usage documentation for information on how to create a venv. If you really want to install system-wide, you can (using sudo).

```
pip install rpymostat
```

Configuration

The RPyMostat Engine is configured solely via environment variables. This is intended to make it simple to run at the command line, as a system service, or in a container.

Usage

Something.

RaspberryPyMostat Architecture

Overview

The architecture of RaspberryPyMostat is made up of four main components:

- Engine - the main control process, which handles the actual data evaluation, schedling and control decisions. This also connects directly to the databases, and provides a ReST API.
- UI - The web interface (desktop and mobile), currently planned to be Flask. This is a standalone "thing", which simply communicates with the main control process via its ReST API.
- Control - One (or more) physical control processes, which receive instructions from the main control process via a ReST API, and control relays (or whatever is needed to drive the actual HVAC equipment).
- Sensor - One or more temperature sensors, which send their results back to the main control process via its ReST API.

While the typical implementation will likely place all of these components on a RaspberryPi (single host deployment), this is by no means a requirement. Each of the three non-main services (web UI, physical control and temperature sensors) are completely independent. They detect the master control process via DNS-SD meaning that they can live on different machines on the same LAN, and find each other without any manual configuration.

All of the separate components communicate with each other over ReST APIs, meaning that temperature inputs, physical control outputs, and web interfaces can be replaced with any third-party code that conforms to the API.

Service Discovery

There are a few options out there for service discovery. I'm only considering ones that are cross-platform and language-agnostic, so this pretty much means DNS-SD or SLP.

python-zeroconf

python-zeroconf appears to be the current winner.

- Native pure-python implementation; no annoying ctypes or external dependencies
- hosted on pypi
- an actual package, complete with tests (TravisCI) and coverage reports
- Little to no logging; I have a branch that fixes this and a few other things
- re-registration of services seems broken (see issue 16) as does responding to requests
- the examples actually run; I got them working on 2 separate machines, but only if the browser was started *before* the registration happened
- some issues around logging (see issue 15)
- updated relatively recently
- Need to see if this will run in Twisted, or if I'd just run it as a separate process or thread...

pybonjour

pybonjour is one option, but it has a lot of drawbacks

- the example code dies for me with mysterious, meaningless exceptions
- not Python3 compatible; there's a fork that is, but its examples don't even work under python3
- no testing
- download isn't actually hosted on pypi
- seems relatively abandoned
- depends on Avahi's bonjour compatibility libs, or Bonjour, depending on platform

Avahi Bindings

Avahi's official bindings

- they're stable, because they're used by Avahi itself
- testing would be a pain, because they require a bunch of libraries like DBUS and Avahi itself
- uses Avahi over DBUS - doesn't do things itself, and requires Avahi to be running
- massive external dependencies, no real package

Other Options

- anyMesh - A multi-platform, decentralized, auto-discover and auto-connect mesh networking and messaging API
- Twisted and UDP multicast?
- python-brisa UPnP framework SSDP Server
- mdns - Python, no docs, no readme

- txbonjour - a Twisted plugin for Bonjour; looks simple, it might be the right thing if it works - based on pybonjour, so probably a no-go

Twisted for RPyMostat

This documents my initial tests for using Twisted as the framework for the main/hub process.

First, a really good basic doc on Deferreds and a not-so-short introduction to Asynchronous programming.

Twisted Basics

- Reactors
- Spawning Processes
- Deferreds; Deferred Reference
- Writing Servers
- Scheduling - "run in X seconds" or "run every N seconds"
- Threading - "most code in Twisted is not thread-safe"
- Application Framework - seems like this might be too baisc for my needs?
- Logging using the logging module, and capturing Twisted's internal messages; note this can block logging

The more I read about this, the more I think Twisted is probably *not* the solution I need (I seem to need *real* threading or multiprocessing, not just async network IO). i.e. see this really important FAQ, Why does it take a long time for data I send with transport.write to arrive at the other side of the connection?.

Third-Party Twisted Modules

- Paisley CouchDB client
- sAsync Async SQLAlchemy
- TxScheduling cron-like scheduling
- txAMQP
- txrdq resizable dispatch queue

ReST API

- we need to serve it nicely (not a horrible hack)
- read/write from database used by other threads (DB tech still unknown; maybe flat files for now?)
- read/write to some shared global memory (or main thread)

Links

- web development with Twisted
- web services with twisted
- Building RESTful, Service-Oriented Architectures with Twisted video and slide deck
- Twisted community code and add-ons
- Klein a web micro-framework
- Going asynchronous: from Flask to Twisted Klein

Klein

Links

- some other projects appear to use klein (in non-trivial ways)
- this blog post was actually VERY helpful

Signals

- Signals or some other sort of notification mechanism

Links

- helga uses smokesignal quite nicely

Scheduling

- Scheduled tasks

Testing

- test-ability (i.e. pytest, possibly something else to test the threading/network)

Links

- pytest-twisted
- Twisted TDD/Trial docs
- TwistedTrial
- Unit Tests in Twisted (internal to Twisted itself)
- some notes on using nose to test Twisted
- Random selection of GitHub projects using pytest-twisted: scrapy, pyrake, snappy, pokerthproto, mcloud
- Interestingly, I can't find *anything* on GitHub that uses pytest-twisted's `pytest.blockon`. `pytest.inlineCallbacks` is used by a number of mcloud tests, jukebox, webmonitor and spiral

rpymostat

rpymostat package

Subpackages

rpymostat.db package

rpymostat.db.**connect_mongodb**(*host*, *port*)
Run *setup_mongodb()*. If that succeeds, connect to MongoDB via txmongo. Return a txmongo ConnectionPool.

Parameters

- **host** (*str*) – host to connect to MongoDB on.
- **port** (*int*) – port to connect to MongoDB on.

Returns MongoDB connection pool

Return type txmongo.connection.ConnectionPool

rpymostat.db.**get_collection**(*db_conn*, *collection_name*)
Return the specified collection object from the database.

Parameters

- **db_conn** (*txmongo.connection.ConnectionPool*) – MongoDB ConnectionPool
- **collection_name** (*str*) – name of the collection to get; should be a constant in this module

Returns txmongo.collection.Collection

rpymostat.db.**setup_mongodb**(*host*, *port*)
Connect synchronously (outside/before the reactor loop) to MongoDB and setup whatever we need. Raise an exception if this fails. This mainly exists to test that the DB is running and accessible before running the reactor loop.

Parameters

- **host** (*str*) – host to connect to MongoDB on.
- **port** (*int*) – port to connect to MongoDB on.

Submodules

rpymostat.db.sensors module

rpymostat.db.sensors.**update_sensor**(**args*, ***kwargs*)
Update data for a single sensor in the database.

Parameters

- **dbconn** (*txmongo.connection.ConnectionPool*) – MongoDB database connection
- **host_id** (*str*) – host_id that the sensor is reporting from

- **sensor_id** (*str*) – unique sensor ID
- **value** (*float*) – sensor reading in degress Celsius
- **sensor_type** (*str*) – description of the type of sensor
- **sensor_alias** (*str*) – human-readable alias for the sensor
- **extra** (*str*) – extra information about the sensor

Returns database record ID

Return type str

rpymostat.engine package

Subpackages

rpymostat.engine.api package

Subpackages

rpymostat.engine.api.v1 package

class rpymostat.engine.api.v1.**APIv1**(*apiserver*, *app*, *dbconn*, *parent_prefix*)
Bases: *rpymostat.engine.site_hierarchy.SiteHierarchy*

Implementation of API v1. All pieces of the v1 API are initialized here, and routes are setup.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

prefix_part = 'v1'

setup_routes()
Setup routes for subparts of the hierarchy.

Submodules

rpymostat.engine.api.v1.sensors module

class rpymostat.engine.api.v1.sensors.**Sensors**(*apiserver*, *app*, *dbconn*, *parent_prefix*)
Bases: *rpymostat.engine.site_hierarchy.SiteHierarchy*

Manages the v1/sensors portion of the API.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

list(*_self*, *request*)
Handle sensor list API endpoint - return a list of known sensors.

This serves the *GET /v1/sensors* endpoint.

Parameters

- **_self** – another reference to `self` sent by Klein
- **request** (instance of `twisted.web.server.Request`) – the Request

prefix_part = 'sensors'

setup_routes()
Setup routes for subparts of the hierarchy.

update(**args*, ***kwargs*)
Handle updating data from a remote sensor.

This serves *PUT /v1/sensors/update* endpoint.

@TODO Handle sensor data update.

Parameters

- **_self** – another reference to `self` sent by Klein
- **request** (instance of `twisted.web.server.Request`) – the Request

rpymostat.engine.api.v1.status module

class rpymostat.engine.api.v1.status.**Status**(*apiserver*, *app*, *dbconn*, *parent_prefix*)
Bases: *rpymostat.engine.site_hierarchy.SiteHierarchy*

Manages the v1/status portion of the API.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

prefix_part = 'status'

setup_routes()
Setup routes for subparts of the hierarchy.

status(**args*, ***kwargs*)
Report on application and dependency status.

This serves *GET /v1/status*

Parameters

- **_self** – another reference to `self` sent by Klein
- **request** (instance of `twisted.web.server.Request`) – the Request

Submodules

rpymostat.engine.apiserver module

class rpymostat.engine.apiserver.**APIServer**(*dbconn=None*)
Bases: object

Main class for the Klein-based API server.

app
Global class attribute pointing to a Klein instance.

handle_root(*_self*, *request*)
root resource (/) request handler. This should only be called by the Kelin app as a route.

This serves the *GET /* endpoint.

@TODO this should return some helpful information, like the server version and a link to the docs, as well as where to obtain the source code and a link to the status page.

Parameters

- **_self** – another reference to self, sent by Klein.
- **request** (instance of twisted.web.server.Request) – the Request

rpymostat.engine.site_hierarchy module

class rpymostat.engine.site_hierarchy.**SiteHierarchy**(*apiserver*, *app*, *dbconn*, *parent_prefix*)
Bases: object

Helper class to implement hierarchical sites in Klein. All engine classes that provide routes must implement this.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 29

_abc_registry = <_weakrefset.WeakSet object>

_parse_json_request(*request*)
Parse a JSON request; return the deserialized request or raise an exception.

Parameters **request** (instance of twisted.web.server.Request) – the request

Returns deserialized request JSON

Return type str

add_route(*func, path=None, methods=['GET']*)
Add a route to app, mapping func (a callable method in this class) to path (under self.prefix). If path is none, it will be mapped directly to self.prefix.

Parameters

- **func** – callable in this class to map to path
- **path** (*str*) – path to map method to
- **methods** (list) – methods allowed for this route

make_prefix(*parent_list*, *prefix_str*)
Given a list of the parent's prefix and our prefix string, construct a new list with our prefix.

Parameters

- **parent_list** (list) – parent's prefix
- **prefix_str** (*str*) – our prefix string

Returns our prefix list

Return type *list*

prefix_list_to_str(*prefix_list*)
Convert a prefix list to a string path prefix.

Parameters **prefix_list** (list) – the list to convert

Returns prefix string

Return type str

prefix_part = 'base'

setup_routes()
Setup all routes for this class. Must be implemented by subclasses.

Submodules

rpymostat.config module

class rpymostat.config.**Config**
Bases: object

RPyMostat configuration. Reads configuration from environmnet variables, sets defaults for anything missing.

_config_vars = {'mongo_port': {'default_value': 27017, 'is_int': True, 'description': 'port number to connect to Mon

_get_from_env()
Build self._config from env vars and defaults (self._config_vars).

Returns effective configuration dict

as_dict
Return the full configuration dictionary, setting names to values.

Returns configuration dict

Return type dict

get(*setting_name*)
Return the effective value for the configuration option setting_name

Parameters **setting_name** (*str*) – the config setting to get

Returns str or int configuration value

get_var_info()
Return information about configuration variables. Returns a dict keyed by setting name. Values are dicts with keys:

•env_var_name - environment variable name for this setting

•description - description of this variable

- `is_int` - boolean, whether the value should be an int or a str
- `default_value` - default value if not present in os.environ

Returns dict describing configuration variables

Return type dict

rpymostat.exceptions module

exception `rpymostat.exceptions.`**`RequestParsingException`**(*message*)
Bases: `exceptions.Exception`

rpymostat.runner module

`rpymostat.runner.`**`main`**(*args=None*)
Run the Engine API server

`rpymostat.runner.`**`parse_args`**(*argv*)
Use Argparse to parse command-line arguments.

Parameters **`argv`** (`list`) – list of arguments to parse (`sys.argv[1:]`)

Returns parsed arguments

Return type `argparse.Namespace`

`rpymostat.runner.`**`set_log_debug`**()
set logger level to DEBUG, and debug-level output format

`rpymostat.runner.`**`set_log_info`**()
set logger level to INFO

`rpymostat.runner.`**`set_log_level_format`**(*level*, *format*)
Set logger level and format.

Parameters

- **`level`** (*`int`*) – logging level; see the `logging` constants.
- **`format`** (*`str`*) – logging formatter format string

`rpymostat.runner.`**`show_config`**(*conf*)
Show configuration variable information.

Parameters **`conf`** (*`Config`*) – config

rpymostat.version module

RPyMostat Engine HTTP API

`GET /`
Simple informational page that returns HTML describing the program and version, where to find the source code, and links to the documentation and status page.

Served by *`handle_root()`*.

Example request:

```
GET / HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – no error

GET /v1/sensors

Return application status information. Currently just returns the text string "Status: Running".

Served by *list()*.

Example request:

```
GET /v1/sensors HTTP/1.1
Host: example.com
```

Example Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

Status Codes

- 200 OK – no errors

PUT /v1/sensors/update

Update current data/readings for sensors from a remote RPyMostat-sensor device.

Served by *update()*.

Example request:

```
PUT /v1/sensors/update HTTP/1.1
Host: example.com

{
    'host_id': 'myhostid',
    'sensors': {
        '1058F50F01080047': {
           'type': 'DS18S20',
           'value': 24.3125,
           'alias': 'some_alias',
           'extra': 'arbitrary string'
        },
        ...
    }
}
```

Sensor Data Objects:

The `sensors` request attribute is itself an object (dict/hash). Keys are globally-unique sensor IDs. The value is an object with the following fields/attributes/keys:

•**type:** *(string)* descriptive type, such as the sensor model

- **value:** *(float/decimal or null)* decimal current temperature reading in degrees Celsius, or null if the current reading cannot be determined.
- **alias:** *(optional; string)* a human-readable alias or name for this sensor, if the system it runs on contains this information. This is not to be confused with the name that RPyMostat maintains for the sensor.
- **extra:** *(optional; string)* arbitrary further information about this sensor, to be included in details about it.

Example Response:

```
HTTP/1.1 202 OK
Content-Type: application/json

{"status": "ok", "ids": [ "id_1", "id_2" ]}
```

Example Response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{"status": "error", "error": "host_id field is missing"}
```

Request JSON Object

- **host_id** – *(string)* the unique identifier of the sending host
- **sensors** – *(object)* array of sensor data objects, conforming to the description above.

Response JSON Object

- **status** – *(string)* the status of the update; `accepted` or `done`
- **id** – *(int)* unique identifier for the update

Status Codes

- 201 Created – update has been made in the database

GET /v1/status

Return application status information (mainly dependent services).

Served by *status()*.

Example request:

```
GET /v1/status HTTP/1.1
Host: example.com
```

Example Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status": true,
    "dependencies": {
        "mongodb": true
    }
}
```

Status Codes

- 200 OK – operational
- 503 Service Unavailable – non-operational

Changelog

x.y.z (YYYY-MM-DD)

- something

RPyMostat Development

To install for development:

1. Fork the RPyMostat repository on GitHub
2. Create a new branch off of master in your fork.

```
$ git clone git@github.com:YOURNAME/RPyMostat.git
$ cd RPyMostat
$ virtualenv . && source bin/activate
$ pip install -r requirements_dev.txt
$ python setup.py develop
```

The git clone you're now in will probably be checked out to a specific commit, so you may want to `git checkout BRANCHNAME`.

Guidelines

- pep8 compliant with some exceptions (see pytest.ini)
- 100% test coverage with pytest (with valid tests)

Testing

Testing is done via pytest, driven by tox.

- testing is as simple as:
  - `pip install tox`
  - `tox`
- If you want to see code coverage: `tox -e cov`
  - this produces two coverage reports - a summary on STDOUT and a full report in the `htmlcov/` directory
- If you want to pass additional arguments to pytest, add them to the tox command line after "–". i.e., for verbose pytext output on py27 tests: `tox -e py27 -- -v`

Acceptance Tests

Acceptance tests run against a real MongoDB. When running locally, they assume that Docker is present and usable, and will pull and run a container from jantman/mongodb24. When running on TravisCI, they will use the mongodb service provided by Travis. The Travis MongoDB service currently runs 2.4.12 as of 2016-06-11, which I'm considering close enough to the Debian 2.4.10 that we're targeting. If Travis upgrades that, we may need to look into alternate ways of running Mongo for the Travis tests.

By default, when run locally, the acceptance tests will start up the MongoDB container when the test session starts, and stop and remove it when the session is over. To leave the container running and reuse it for further test sessions, export the `LEAVE_MONGO_RUNNING` environment variable.

Release Checklist

1. Open an issue for the release; cut a branch off master for that issue.
2. Confirm that there are CHANGES.rst entries for all major changes.
3. Ensure that Travis tests passing in all environments.
4. Ensure that test coverage is no less than the last release (ideally, 100%).
5. Increment the version number in RPyMostat/version.py and add version and release date to CHANGES.rst, then push to GitHub.
6. Confirm that README.rst renders correctly on GitHub.
7. Upload package to testpypi, confirm that README.rst renders correctly.
    - Make sure your ~/.pypirc file is correct
    - `python setup.py register -r https://testpypi.python.org/pypi`
    - `python setup.py sdist upload -r https://testpypi.python.org/pypi`
    - Check that the README renders at https://testpypi.python.org/pypi/rpymostat
8. Create a pull request for the release to be merge into master. Upon successful Travis build, merge it.
9. Tag the release in Git, push tag to GitHub:
    - tag the release. for now the message is quite simple: `git tag -a vX.Y.Z -m 'X.Y.Z released YYYY-MM-DD'`
    - push the tag to GitHub: `git push origin vX.Y.Z`
11. Upload package to live pypi:
    - `python setup.py sdist upload`
10. make sure any GH issues fixed in the release were closed.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

License

RPyMostat is licensed under the GNU Affero General Public License, version 3 or later.

HTTP Routing Table

/

/v1

Python Module Index

r

Index

Symbols

A

C

G

H

L

M

P

R

S

U