AppDaemon Documentation

Release 2.1.12

Andrew Cockburn

Jan 14, 2018

Contents

1 Installation 3

2 AppDaemon Tutorial 11

3 Appdaemon with Docker 15

4 Writing AppDaemon Apps 21

5 AppDaemon API Reference 47

6 Dashboard Install and Configuration 85

7 Dashboard Creation 89

8 HADashboard Widget Development 121

9 Change Log 139

AppDaemon is a loosely coupled, multithreaded, sandboxed python execution environment for writing automation apps for Home Assistant home automation software. As of release 2.0.0 it also provides a configurable dashboard (HADashboard) suitable for wall mounted tablets.

Contents:

CHAPTER 1

Installation

Installation is either by pip3 or Docker. There is also an official hass.io build.

Note: Windows and Raspbian users should check the environment specific section at the end of this doc for additional information.

1.1 Install and Run using Docker

Follow the instructions in the Docker Tutorial

1.2 Install Using pip3

Before running `AppDaemon` you will need to install the package:

```
$ sudo pip3 install appdaemon
```

1.3 Install Using hass.io

There is an official hass.io addon for AppDaemon maintained by vkorn. Instructions to install AppDaemon this way can be found here

1.4 Configuration

When you have appdaemon installed by either method you are ready to start working on the appdaemon.yaml file. For docker users, you will already have a skeleton to work with. For pip users, you need to create a configuration directory somewhere (e.g. `/home/homeassistant/conf`) and create a file in there called `appdaemon.yaml`.

Your initial file should look something like this:

```
AppDaemon:
  logfile: STDOUT
  errorfile: STDERR
  logsize: 100000
  log_generations: 3
  threads: 10
  cert_path: <path/to/root/CA/cert>
  cert_verify: True
  time_zone: <time zone>
  api_port: 5000
  api_key: !secret api_key
  api_ssl_certificate: <path/to/root/CA/cert>
  api_ssl_key: <path/to/root/CA/key>
HASS:
  ha_url: <some_url>
  ha_key: <some key>
```

- `ha_url` is a reference to your home assistant installation and must include the correct port number and scheme (`http://` or `https://` as appropriate)
- `ha_key` should be set to your key if you have one, otherwise it can be removed.
- `logfile` (optional) is the path to where you want `AppDaemon` to keep its main log. When run from the command line this is not used -log messages come out on the terminal. When running as a daemon this is where the log information will go. In the example above I created a directory specifically for AppDaemon to run from, although there is no reason you can't keep it in the `appdaemon` directory of the cloned repository. If `logfile = STDOUT`, output will be sent to stdout instead of stderr when running in the foreground, if not specified, output will be sent to STDOUT.
- `errorfile` (optional) is the name of the logfile for errors - this will usually be errors during compilation and execution of the apps. If `errorfile = STDERR` errors will be sent to stderr instead of a file, if not specified, output will be sent to STDERR.
- `log_size` (optional) is the maximum size a logfile will get to before it is rotated if not specified, this will default to 1000000 bytes.
- `log_generations` (optional) is the number of rotated logfiles that will be retained before they are overwritten if not specified, this will default to 3 files.
- `threads` - the number of dedicated worker threads to create for running the apps. Note, this will bear no resembelance to the number of apps you have, the threads are re-used and only active for as long as required to run a particular callback or initialization, leave this set to 10 unless you experience thread starvation
- `cert_path` (optional) - path to root CA cert directory for HASS - use only if you are using self signed certs.
- `cert_verify` (optional) - flag for cert verification for HASS - set to `False` to disable verification on self signed certs.
- `time_zone` (optional) - timezone for AppDaemon to use. If not specified, AppDaemon will query the timezone from Home Assistant
- `api_port` (optional) - Port the AppDaemon RESTFul API will liten on. If not specified, the RESTFul API will be turned off
- `api_key` (optional) - adds the requirement for AppDaemon API calls to provide a key in the header of a request
- `api_ssl_certificate` (optional) - certificate to use when running the API over SSL
- `api_ssl_key` (optional) - key to use when running the API over SSL

Optionally, you can place your apps in a directory other than under the config directory using the `app_dir` directive.

e.g.:

```
app_dir: /etc/appdaemon/apps
```

1.4.1 Secrets

AppDaemon supports the use of secrets in the configuration file (YAML only), to allow separate storage of sensitive information such as passwords. For this to work, AppDaemon expects to find a file called secrets.yaml in the configuration directory with a simple list of all the secrets. The secrets can be referred to using a !secret value in the configuration file.

An example secrets.yaml might look like this:

```
home_assistant_key_key: password123
appdaemon_key: password456
```

The secrets can then be referred to as follows:

```
AppDaemon:
  api_key: !secret appdaemon_key
  threads: '10'
HASS:
  ha_key: !secret home_assistant_key
  ha_url: http://192.168.1.20:8123
```

1.4.2 Configuring a Test App

To start configuring Apps, we need to create a new *apps.yaml* file in the same directory as appdaemon.yaml. To start, we can add an entry for the Hello World App like this:

```
hello_world:
  module: hello
  class: HelloWorld
```

App configuration is fully described in the API doc.

To add an initial test app to match the configuration above, we need to first create an `apps` subdirectory under the conf directory. Then create a file in the apps directory called `hello.py`, and paste the following into it using your favorite text editor:

```
import appdaemon.appapi as appapi

#
# Hello World App
#
# Args:
#

class HelloWorld(appapi.AppDaemon):

  def initialize(self):
     self.log("Hello from AppDaemon")
     self.log("You are now ready to run Apps!")
```

With this app in place we will be able to test the App part of AppDaemon when we first run it.

1.4.3 Configuring the Dashboard

Configuration of the dashboard component (HADashboard) is described separately in the Dashboard doc

1.5 Example Apps

There are a number of example apps under `conf/examples` in the git repository, and the `conf/examples.yaml` file gives sample parameters for them.

1.6 Running

1.6.1 Docker

Assuming you have set the config up as described in the tutotial for Docker, you should see the logs output as follows:

```
$ docker logs appdaemon
2016-08-22 10:08:16,575 INFO Got initial state
2016-08-22 10:08:16,576 INFO Loading Module: /export/hass/appdaemon_test/conf/apps/
↪hello.py
2016-08-22 10:08:16,578 INFO Loading Object hello_world using class HelloWorld from␣
↪module hello
2016-08-22 10:08:16,580 INFO Hello from AppDaemon
2016-08-22 10:08:16,584 INFO You are now ready to run Apps!
```

Note that for Docker, the error and regular logs are combined.

1.6.2 PIP3

You can run AppDaemon from the command line as follows:

```
$ appdaemon -c /home/homeassistant/conf
```

If all is well, you should see something like the following:

```
$ appdaemon -c /home/homeassistant/conf
2016-08-22 10:08:16,575 INFO Got initial state
2016-08-22 10:08:16,576 INFO Loading Module: /home/homeassistant/conf/apps/hello.py
2016-08-22 10:08:16,578 INFO Loading Object hello_world using class HelloWorld from␣
↪module hello
2016-08-22 10:08:16,580 INFO Hello from AppDaemon
2016-08-22 10:08:16,584 INFO You are now ready to run Apps!
```

1.7 AppDaemon arguments

```
usage: appdaemon [-h] [-c CONFIG] [-p PIDFILE] [-t TICK] [-s STARTTIME]
                 [-e ENDTIME] [-i INTERVAL]
                 [-D {DEBUG,INFO,WARNING,ERROR,CRITICAL}] [-v] [-d]

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        full path to config diectory
  -p PIDFILE, --pidfile PIDFILE
                        full path to PID File
  -t TICK, --tick TICK  time in seconds that a tick in the schedular lasts
  -s STARTTIME, --starttime STARTTIME
                        start time for scheduler <YYYY-MM-DD HH:MM:SS>
  -e ENDTIME, --endtime ENDTIME
                        end time for scheduler <YYYY-MM-DD HH:MM:SS>
  -i INTERVAL, --interval INTERVAL
                        multiplier for scheduler tick
  -D {DEBUG,INFO,WARNING,ERROR,CRITICAL}, --debug {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        debug level
  -v, --version         show program's version number and exit
  -d, --daemon          run as a background process
```

-c is the path to the configuration directory. If not specified, AppDaemon will look for a file named `appdaemon.cfg` first in `~/.homeassistant` then in `/etc/appdaemon`. If the directory is not specified and it is not found in either location, AppDaemon will raise an exception. In addition, AppDaemon expects to find a dir named `apps` immediately subordinate to the config directory.

-d and -p are used by the init file to start the process as a daemon and are not required if running from the command line.

-D can be used to increase the debug level for internal AppDaemon operations as well as apps using the logging function.

The -s, -i, -t and -s options are for the Time Travel feature and should only be used for testing. They are described in more detail in the API documentation.

1.8 Legacy Configuration

AppDaemon also currently supports a legacy `ini` style of configuration and it is shown here for backward compatibility. It is recommended that you move to the YAML format using the provided tool. When using the legacy configuration style, there are no `HASS` or `HADashboard` sections - the associated directives all go in the `AppDaemon` section.

```
[AppDaemon]
ha_url = <some_url>
ha_key = <some key>
logfile = STDOUT
errorfile = STDERR
threads = 10
cert_path = <path/to/root/CA/cert>
cert_verify = True
# Apps
[hello_world]
module = hello
class = HelloWorld
```

If you want to move from the legacy `ini` style of configuration to YAML, AppDaemon is able to do this for you. Just run AppDaemon providing the configuration directory using the -c option as usual and specify the –convertcfg flag. From the command line run:

```
$ appdaemon -c YOUR_CONFIG_DIR --convertcfg
Converting /etc/appdaemon/appdaemon.cfg to /etc/appdaemon/appdaemon.yaml
$
```

AppDaemon should correctly figure out where the file is to convert form your existing configuration. After conversion, the new YAML file will be used in preference to the old ini file, which can then be removed if desired.

Note: any lines in the ini file that are commented out, whether actual comments of lines that are not active, will not be converted. Note 2: Docker users will unfortunately need to perform the conversion manually.

1.9 Starting At Reboot

To run `AppDaemon` at reboot, you can set it up to run as a systemd service as follows.

1.9.1 Add Systemd Service (appdaemon@appdaemon.service)

First, create a new file using vi:

```
$ sudo vi /etc/systemd/system/appdaemon@appdaemon.service
```

Add the following, making sure to use the correct full path for your config directory. Also make sure you edit the `User` to a valid user to run AppDaemon, usually the same user as you are running Home Assistant with is a good choice.

```
[Unit]
Description=AppDaemon
After=home-assistant@homeassistant.service
[Service]
Type=simple
User=hass
ExecStart=/usr/local/bin/appdaemon -c <full path to config directory>
[Install]
WantedBy=multi-user.target
```

The above should work for hasbian, but if your homeassistant service is named something different you may need to change the `After=` lines to reflect the actual name.

1.9.2 Activate Systemd Service

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable appdaemon@appdaemon.service --now
```

Now AppDaemon should be up and running and good to go.

1.10 Operation

Since AppDaemon under the covers uses the exact same APIs as the frontend UI, you typically see it react at about the same time to a given event. Calling back to Home Assistant is also pretty fast especially if they are running on the same machine. In action, observed latency above the built in automation component is usually sub-second.

1.11 Updating AppDaemon

To update AppDaemon after new code has been released, just run the following command to update your copy:

```
$ sudo pip3 install --upgrade appdaemon
```

If you are using docker, refer to the steps in the tutorial.

1.12 Windows Support

AppDaemon runs under windows and has been tested with the official 3.5.2 release of python. There are a couple of caveats however:

- The `-d` or `--daemonize` option is not supported owing to limitations in the Windows implementation of Python.
- Some internal diagnostics are disabled. This is not user visible but may hamper troubleshooting of internal issues if any crop up

AppDaemon can be installed exactly as per the instructions for every other version using pip3.

1.13 Windows Under the Linux Subsystem

Windows 10 now supports a full Linux bash environment that is capable of running Python. This is essentially an Ubuntu distribution and works extremely well. It is possible to run AppDaemon in exactly the same way as for Linux distributions, and none of the above Windows Caveats apply to this version. This is the reccomended way to run AppDaemon in a Windows 10 and later environment.

1.14 Raspbian

Some users have reported a requirement to install a couple of packages prior to installing AppDaemon with the pip3 method:

```
$ sudo apt-get install python-dev
$ sudo apt-get install libffi-dev
```

CHAPTER 2

AppDaemon Tutorial

I have been working on a new subsystem to complement Home Assistant's Automation and Scripting components. `AppDaemon` is a python daemon that consumes events from Home Assistant and feeds them to snippets of python code called "Apps". An App is a Python class that is instantiated possibly multiple times from AppDaemon and registers callbacks for various system events. It is also able to inspect and set state and call services. The API provides a rich environment suited to home automation tasks that can also leverage all the power of Python.

2.1 Another Take on Automation

If you haven't yet read Paulus' excellent Blog entry on Perfect Home Automation I would encourage you to take a look. As a veteran of several Home Automation systems with varying degrees success, it was this article more than anything else that convinced me that Home Assistant had the right philosophy behind it and was on the right track. One of the most important points made is that being able to control your lights from your phone, 9 times out of 10 is harder than using a lightswitch - where Home Automation really comes into its own is when you start removing the need to use a phone or the switch - the "Automation" in Home Automation. A surprisingly large number of systems out there miss this essential point and have limited abilities to automate anything which is why a robust and open system such as Home Assistant is such an important part of the equation in bring this all together in the vast and chaotic ecosystem that is the "Internet of Things".

So given the importance of Automation, what should Automation allow us to do? I am a pragmatist at heart so I judge individual systems by the ease of accomplishing a few basic but representative tasks:

- Can the system respond to presence or absence of people?
- Can I turn a light on at Sunset +/- a certain amount of time?
- Can I arrive home in light or dark and have the lights figure out if they should be on or off?
- As I build my system out, can I get the individual pieces to co-operate and use and re-use (potentially complex) logic to make sure everything works smoothly?
- Is it open and expandable?
- Does it run locally without any reliance on the cloud?

In my opinion, Home Assistant accomplishes the majority of these very well with a combination of Automations, Scripts and Templates, and it's Restful API.

So why `AppDaemon`? AppDaemon is not meant to replace Home Assistant Automations and Scripts, rather complement them. For a lot of things, automations work well and can be very succinct. However, there is a class of more complex automations for which they become harder to use, and appdeamon then comes into its own. It brings quite a few things to the table:

- New paradigm - some problems require a procedural and/or iterative approach, and `AppDaemon` Apps are a much more natural fit for this. Recent enhancements to Home Assistant scripts and templates have made huge strides, but for the most complex scenarios, Apps can do things that Automations can't
- Ease of use - AppDaemon's API is full of helper functions that make programming as easy and natural as possible. The functions and their operation are as "Pythonic" as possible, experienced Python programmers should feel right at home.
- Reuse - write a piece of code once and instantiate it as an app as many times as you need with different parameters e.g. a motion light program that you can use in 5 different places around your home. The code stays the same, you just dynamically add new instances of it in the config file
- Dynamic - AppDaemon has been designed from the start to enable the user to make changes without requiring a restart of Home Assistant, thanks to it's loose coupling. However, it is better than that - the user can make changes to code and AppDaemon will automatically reload the code, figure out which Apps were using it and restart them to use the new code with out the need to restart `AppDaemon` itself. It is also possible to change parameters for an individual or multiple apps and have them picked up dynamically, and for a final trick, removing or adding apps is also picked up dynamically. Testing cycles become a lot more efficient as a result.
- Complex logic - Python's If/Else constructs are clearer and easier to code for arbitrarily complex nested logic
- Durable variables and state - variables can be kept between events to keep track of things like the number of times a motion sensor has been activated, or how long it has been since a door opened
- All the power of Python - use any of Python's libraries, create your own modules, share variables, refactor and re-use code, create a single app to do everything, or multiple apps for individual tasks - nothing is off limits!

It is in fact a testament to Home Assistant's open nature that a component like `AppDaemon` can be integrated so neatly and closely that it acts in all ways like an extension of the system, not a second class citizen. Part of the strength of Home Assistant's underlying design is that it makes no assumptions whatever about what it is controlling or reacting to, or reporting state on. This is made achievable in part by the great flexibility of Python as a programming environment for Home Assistant, and carrying that forward has enabled me to use the same philosophy for `AppDaemon` - it took surprisingly little code to be able to respond to basic events and call services in a completely open ended manner - the bulk of the work after that was adding additonal functions to make things that were already possible easier.

2.2 How it Works

The best way to show what AppDaemon does is through a few simple examples.

2.2.1 Sunrise/Sunset Lighting

Lets start with a simple App to turn a light on every night fifteen minutes (900 seconds) before sunset and off every morning at sunrise. Every App when first started will have its `initialize()` function called which gives it a chance to register a callback for AppDaemons's scheduler for a specific time. In this case we are using `run_at_sunrise()` and `run_at_sunset()` to register 2 separate callbacks. The named argument `offset` is the number of seconds offset from sunrise or sunset and can be negative or positive (it defaults to zero). For complex intervals it can be convenient to use Python's `datetime.timedelta` class for calculations. In the example below, when sunrise or just before sunset occurs, the appropriate callback function, `sunrise_cb()` or `before_sunset_cb()` is called

which then makes a call to Home Assistant to turn the porch light on or off by activating a scene. The variables `args["on_scene"]` and `args["off_scene"]` are passed through from the configuration of this particular App, and the same code could be reused to activate completely different scenes in a different version of the App.

```
import homeassistant.appapi as appapi

class OutsideLights(appapi.AppDaemon):

  def initialize(self):
    self.run_at_sunrise(self.sunrise_cb)
    self.run_at_sunset(self.before_sunset_cb, offset=-900)

  def sunrise_cb(self, kwargs):
    self.turn_on(self.args["off_scene"])

  def before_sunset_cb(self, kwargs):
    self.turn_on(self.args["on_scene"])
```

This is also fairly easy to achieve with Home Assistant automations, but we are just getting started.

2.2.2 Motion Light

Our next example is to turn on a light when motion is detected and it is dark, and turn it off after a period of time. This time, the `initialize()` function registers a callback on a state change (of the motion sensor) rather than a specific time. We tell AppDaemon that we are only interested in state changes where the motion detector comes on by adding an additional parameter to the callback registration - `new = "on"`. When the motion is detected, the callack function `motion()` is called, and we check whether or not the sun has set using a built-in convenience function: `sun_down()`. Next, we turn the light on with `turn_on()`, then set a timer using `run_in()` to turn the light off after 60 seconds, which is another call to the scheduler to execute in a set time from now, which results in `AppDaemon` calling `light_off()` 60 seconds later using the `turn_off()` call to actually turn the light off. This is still pretty simple in code terms:

```
import homeassistant.appapi as appapi

class FlashyMotionLights(appapi.AppDaemon):

  def initialize(self):
    self.listen_state(self.motion, "binary_sensor.drive", new = "on")

  def motion(self, entity, attribute, old, new, kwargs):
    if self.sun_down():
      self.turn_on("light.drive")
      self.run_in(self.light_off, 60)

  def light_off(self, kwargs):
    self.turn_off("light.drive")
```

This is starting to get a little more complex in Home Assistant automations requiring an Automation rule and two separate scripts.

Now lets extend this with a somewhat artificial example to show something that is simple in AppDaemon but very difficult if not impossible using automations. Lets warn someone inside the house that there has been motion outside by flashing a lamp on and off 10 times. We are reacting to the motion as before by turning on the light and setting a timer to turn it off again, but in addition, we set a 1 second timer to run `flash_warning()` which when called, toggles the inside light and sets another timer to call itself a second later. To avoid re-triggering forever, it keeps a count of how many times it has been activated and bales out after 10 iterations.

```
import homeassistant.appapi as appapi

class MotionLights(appapi.AppDaemon):

  def initialize(self):
    self.listen_state(self.motion, "binary_sensor.drive", new = "on")

  def motion(self, entity, attribute, old, new, kwargs):
    if self.self.sun_down():
      self.turn_on("light.drive")
      self.run_in(self.light_off, 60)
      self.flashcount = 0
      self.run_in(self.flash_warning, 1)

  def light_off(self, kwargs):
    self.turn_off("light.drive")

  def flash_warning(self, kwargs):
    self.toggle("light.living_room")
    self.flashcount += 1
    if self.flashcount < 10:
      self.run_in(self.flash_warning, 1)
```

Of course if I wanted to make this App or its predecessor reusable I would have provide parameters for the sensor, the light to activate on motion, the warning light and even the number of flashes and delay between flashes.

In addition, Apps can write to `AppDaemon`'s logfiles, and there is a system of constraints that allows yout to control when and under what circumstances Apps and callbacks are active to keep the logic clean and simple.

2.3 Final Thoughts

I have spent the last few weeks moving all of my (fairly complex) automations over to `AppDaemon` and so far it is working very reliably.

Some people will maybe look at all of this and say "what use is this, I can already do all of this", and that is fine, as I said this is an alternative not a replacement, but I am hopeful that for some users this will seem a more natural, powerful and nimble way of building potentially very complex automations.

If this has whet your appetite, feel free to give it a try. You can find installation instructions, here, including full installation instructions, an API reference, and a number of fully fleshed out examples.

Happy Automating!

CHAPTER 3

Appdaemon with Docker

A quick tutorial to Appdaemon with Docker

3.1 About Docker

Docker is a popular application container technology. Application containers allow an application to be built in a known-good state and run totally independant of other applications. This makes it easier to install complex software and removes concerns about application dependency conflicts. Containers are powerful, however they require abstractions that can sometimes be confusing.

This guide will help you get the Appdaemon Docker image running and hopefully help you become more comfortable with using Docker. There are multiple ways of doing some of these steps which are removed for the sake of keeping it simple. As your needs change, just remember there's probably a way to do what you want!

3.2 Prereqs

This guide assumes:

- You already have Docker installed. If you still need to do this, follow the Docker Installation documentation
- You have Home Assistant up and running
- You are comfortable with some tinkering. This is a pre-req for Appdaemon too!

3.3 Testing your System

Our first step will be to verify that we can get Appdaemon running on our machine, which tests that we can successfully "pull" (download) software from Docker Hub, execute it, and get output that Appdaemon is working. We will worry about our persistent (normal) configuration later.

Before you start, you need to know the following:

- HA_URL: The URL of your running Home Assistant, in the form of `http://[name]:[port]`. Port is usually 8123.
- HA_KEY: If your Home Assistant requires an API key, you'll need that

Now, on your Docker host, for linux users, run the following command, substituting the values above in the quotes below. (Note, if you do not need an HA_KEY, you can omit the entire -e HA_KEY line)

```
docker run --rm -it -p 5050:5050 \
  -e HA_URL="<your HA_URL value>" \
  -e HA_KEY="<your HA_KEY value>" \
  -e DASH_URL="http://$HOSTNAME:5050" \
  acockburn/appdaemon:latest
```

You should see some download activity the first time you run this as it downloads the latest Appdaemon image. After that is downloaded, Docker will create a container based on that image and run it. It will automatically delete itself when it exits, since right now we are just testing.

You will see Appdaemon's output appear on your screen, and you should look for lines like these being output:

Appdaemon successfully connected to Home Assistant

```
2017-04-01 14:26:48.361140 INFO Connected to Home Assistant 0.40.0
```

The 'apps' capability of Appdaemon is working, running the example Hello World app

```
2017-04-01 14:26:48.330084 INFO hello_world: Hello from AppDaemon
2017-04-01 14:26:48.333040 INFO hello_world: You are now ready to run Apps!
```

The 'dashboard' capability of Appdaemon has started.

```
2017-04-01 14:26:48.348260 INFO HADashboard Started
2017-04-01 14:26:48.349135 INFO Listening on ('0.0.0.0', 5050)
```

Now open up a web browser, and browse to http://:5050. You should see the "Welcome to HADashboard for Home Assistant" screen and see the Hello dashboard is available.

If all of these checks work, congratulations! Docker and Appdaemon are working on your system! Hit Control-C to exit the container, and it will clean up and return to the command line. It's almost as if nothing happened... :)

3.4 Persistent Configuration

In Docker, containers (the running application) are considered ephemeral. Any state that you want to be able to preserve must be stored outside of the container so that the container can be disposed and recreated at any time. In the case of Appdaemon, this means you would be concerned about your `conf` folder.

The first step is to create a location on your filesystem to store the `conf` folder. It does not matter where this is, some people like to store it in the same location as Home Assistant. I like to keep a folder structure under `/docker` on my systems, so we can simply do something like:

```
mkdir -p /docker/appdaemon/conf
```

Next, we will run a container again, omiting the `--rm -it` parameters and adding `-d` so that it stays background and doesn't disappear when it exits. We will also add `--restart=always` so that the container will auto-start on system boot and restart on failures, and lastly specify our `conf` folder location. Note that the folder path must be fully qualified and not relative.

```
docker run --name=appdaemon -d -p 5050:5050 \
  --restart=always \
  -e HA_URL="<your HA_URL value>" \
  -e HA_KEY="<your HA_KEY value>" \
  -e DASH_URL="http://$HOSTNAME:5050" \
  -v <your_conf_folder>:/conf \
  acockburn/appdaemon:latest
```

I would suggest documenting the command line above in your notes, so that you have it as a reference in the future for rebuilding and upgrading. If you back up your command line, as well as your `conf` folder, you can trivially restore Appdaemon on another machine or on a rebuild!

If your `conf` folder is brand new, the Appdaemon Docker will copy the default configuration files into this folder. If there are already configuration files, it will not overwrite them. Double check that the files are there now.

You are now ready to start working on your Appdaemon configurations!

At this point forward, you can edit configurations on your `conf` folder and Appdaemon will load them see the AppDaemon Installation page for full instrctions on AppDaemon configuration. Have fun!

3.5 Viewing Appdaemon Log Output

You can view the output of your Appdaemon with this command:

```
docker logs appdaemon
```

If you'd like to tail the latest output, try this:

```
docker logs -f --tail 20 appdaemon
```

3.6 Upgrading Appdaemon

Upgrading with Docker really doesn't exist in the same way as with non-containerized apps. Containers are considered ephemeral and are an instance of a base, known-good application image. Therefore the process of upgrading is simply disposing of the old version, grabbing a newer version of the application image and starting up a new container with the new version's image. Since the the persistent state (`conf`) was kept, it is effectively an upgrade.

(It is possible to get into downgrades and multiple versions, however in this guide we are keeping it simple!)

Run the following commands:

```
docker stop appdaemon
docker rm appdaemon
docker pull acockburn/appdaemon:latest
docker run --name=appdaemon -d -p 5050:5050 \
  --restart=always \
  -e HA_URL="<your HA_URL value>" \
  -e HA_KEY="<your HA_KEY value>" \
  -e DASH_URL="http://$HOSTNAME:5050" \
  -v <your_conf_folder>:/conf \
  acockburn/appdaemon:latest
```

3.7 Controlling the Appdaemon Conter

To restart Appdaemon:

```
docker restart appdaemon
```

To stop Appdaemon:

```
docker stop appdaemon
```

To start Appdaemon back up after stopping:

```
docker start appdaemon
```

To check the running state, run the following and look at the 'STATUS' column:

```
docker ps -a
```

3.8 Running with Appdaemon Debug

If you need to run Appdaemon with Debug, it may be easiest to stop your normal appdaemon and run a temporary container with the debug flag set. This presumes you already have a configured `conf` folder you are debugging, so we don't need to pass the HA/DASH variables into the container.

Run the following commands:

```
docker stop appdaemon
docker run --rm -it -p 5050:5050 \
  -v <your_conf_folder>:/conf \
  -e EXTRA_CMD="-D DEBUG" \
  acockburn/appdaemon:latest
```

Once you are done with the debug, start the non-debug container back up:

```
docker start appdaemon
```

3.9 Timezones

Some users have reported issues with the Docker container running in different timezones to the host OS - this is obviously problematic for any of the scheduler functions. Adding the following to the Docker command line has helped for some users:

```
-v /etc/localtime:/etc/localtime:ro
```

3.10 Home Assistant SSL

If your Home Assistant is running with self-signed certificates, you will want to point to the location of the certificate files as part of the container creation process. Add `-v <your_cert_path>:/certs` to the `docker run` command line

3.11 Removing Appdaemon

If you no longer want to use Appdaemon :(, use the following commands:

```
docker kill appdaemon
docker rm appdaemon
docker rmi acockburn/appdaemon:latest
```

You can delete the `conf` folder if you wish at this time too. Appdaemon is now completely removed.

CHAPTER 4

Writing AppDaemon Apps

AppDaemon is a loosely coupled, sandboxed, multi-threaded Python execution environment for writing automation apps for Home Assistant home automation software. It is intended to complement the Automation and Script components that Home Assistant currently offers.

4.1 Examples

Example apps that showcase most of these functions are available in the AppDaemon repository:

Apps

4.2 Anatomy of an App

Automations in AppDaemon are performed by creating a piece of code (essentially a Python Class) and then instantiating it as an Object one or more times by configuring it as an App in the configuration file. The App is given a chance to register itself for whatever events it wants to subscribe to, and AppDaemon will then make calls back into the Object's code when those events occur, allowing the App to respond to the event with some kind of action.

The first step is to create a unique file within the apps directory (as defined in the `AppDaemon` section of configuration file - see The Installation Page for further information on the configuration of AppDaemon itself). This file is in fact a Python module, and is expected to contain one or more classes derived from the supplied `AppDaemon` class, imported from the supplied `appdaemon.appapi` module. The start of an app might look like this:

```
import appdaemon.appapi as appapi

class MotionLights(appapi.AppDaemon):
```

When configured as an app in the config file (more on that later) the lifecycle of the App begins. It will be instantiated as an object by AppDaemon, and immediately, it will have a call made to its `initialize()` function - this function must appear as part of every app:

```
def initialize(self):
```

The initialize function allows the app to register any callbacks it might need for responding to state changes, and also any setup activities. When the `initialize()` function returns, the App will be dormant until any of its callbacks are activated.

There are several circumstances under which `initialize()` might be called:

- Initial start of AppDaemon
- Following a change to the Class code
- Following a change to the module parameters
- Following initial configuration of an app
- Following a change in the status of Daylight Saving Time
- Following a restart of Home Assistant

In every case, the App is responsible for recreating any state it might need as if it were the first time it was ever started. If `initialize()` is called, the app can safely assume that it is either being loaded for the first time, or that all callbacks and timers have been cancelled. In either case, the App will need to recreate them. Depending upon the application, it may be desirable for the App to establish a state, such as whether or not a particular light is on, within the `initialize()` function to ensure that everything is as expected or to make immediate remedial action (e.g., turn off a light that might have been left on by mistake when the app was restarted).

After the `initialize()` function is in place, the rest of the app consists of functions that are called by the various callback mechanisms, and any additional functions the user wants to add as part of the program logic. Apps are able to subscribe to two main classes of events:

- Scheduled Events
- State Change Events

These, along with their various subscription calls and helper functions, will be described in detail in later sections.

Optionally, a class can add a `terminate()` function. This function will be called ahead of the reload to allow the class to perform any tidy up that is necessary.

WARNING: Unlike other types of callback, calls to `initialize()` and `terminate()` are synchronous to AppDaemon's management code to ensure that initialization or cleanup is completed before the App is loaded or reloaded. This means that any significant delays in the `terminate()` code could have the effect of hanging AppDaemon for the duration of that code - this should be avoided.

To wrap up this section, here is a complete functioning App (with comments):

```
import appdaemon.appapi as appapi
import datetime

 Declare Class
class NightLight(appapi.AppDaemon):
  #initialize() function which will be called at startup and reload
  def initialize(self):
    # Create a time object for 7pm
    time = datetime.time(19, 00, 0)
    # Schedule a daily callback that will call run_daily() at 7pm every night
    self.run_daily(self.run_daily_callback, time)

   # Our callback function will be called by the scheduler every day at 7pm
  def run_daily_callback(self, kwargs):
```

```
    # Call to Home Assistant to turn the porch light on
    self.turn_on("light.porch")
```

To summarize - an App's lifecycle consists of being initialized, which allows it to set one or more states and/or schedule callbacks. When those callbacks are activated, the App will typically use one of the Service Calling calls to effect some change to the devices of the system and then wait for the next relevant state change. Finally, if the App is reloaded, there is a call to its `terminate()` function if it exists. That's all there is to it!

4.3 About the API

The implementation of the API is located in the AppDaemon class that Apps are derived from. The code for the functions is therefore available to the App simply by invoking the name of the function from the object namespace using the `self` keyword, as in the above examples. `self.turn_on()` for example is just a method defined in the parent class and made available to the child. This design decision was made to simplify some of the implementation and hide passing of unnecessary variables during the API invocation.

4.4 Configuration of Apps

Apps are configured by specifying new sections in the app configuration file - `apps.yaml`. The name of the section is the name the App is referred to within the system in log files etc. and must be unique.

To configure a new App you need a minimum of two directives:

- `module` - the name of the module (without the `.py`) that contains the class to be used for this App
- `class` - the name of the class as defined within the module for the APPs code

Although the section/App name must be unique, it is possible to re-use a class as many times as you want, and conversely to put as many classes in a module as you want. A sample definition for a new App might look as follows:

```
newapp:
  module: new
  class: NewApp
```

When AppDaemon sees the following configuration it will expect to find a class called `NewApp` defined in a module called `new.py` in the apps subdirectory. Apps can be placed at the root of the Apps directory or within a subdirectory, an arbitrary depth down - wherever the App is, as long as it is in some subdirectory of the Apps dir, or in the Apps dir itself, AppDaemon will find it. There is no need to include information about the path, just the name of the file itself (without the `.py`) is sufficient. If names in the subdirectories overlap, AppDir will pick one of them but the exact choice it will make is undefined.

When starting the system for the first time or when reloading an App or Module, the system will log the fact in it's main log. It is often the case that there is a problem with the class, maybe a syntax error or some other problem. If that is the case, details will be output to the error log allowing the user to remedy the problem and reload.

4.5 Steps to writing an App

1. Create the code in a new or shared module by deriving a class from AppDaemon, add required callbacks and code
2. Add the App to the app configuration file
3. There is no number 3

4.6 Reloading Modules and Classes

Reloading of modules is automatic. When the system spots a change in a module, it will automatically reload and recompile the module. It will also figure out which Apps were using that Module and restart them, causing their `terminate()` functions to be called if they exist, all of their existing callbacks to be cleared, and their `initialize()` function to be called.

The same is true if changes are made to an App's configuration - changing the class, or arguments (see later) will cause that app to be reloaded in the same way. The system is also capable of detecting if a new app has been added, or if one has been removed, and it will act appropriately, starting the new app immediately and removing all callbacks for the removed app.

The suggested order for creating a new App is to add the module code first and work until it compiles cleanly, and only then add an entry in the configuration file to actually run it. A good workflow is to continuously monitor the error file (using `tail -f` on Linux for instance) to ensure that errors are seen and can be remedied.

4.7 Passing Arguments to Apps

There wouldn't be much point in being able to run multiple versions of an App if there wasn't some way to instruct them to do something different. For this reason it is possible to pass any required arguments to an App, which are then made available to the object at runtime. The arguments themselves can be called anything (apart from `module` or `class`) and are simply added into the section after the 2 mandatory directives like so:

```
MyApp:
  module: myapp
  class: MyApp
  param1: spam
  param2: eggs
```

Within the Apps code, the 2 parameters (as well as the module and class) are available as a dictionary called `args`, and accessed as follows:

```
param1 = self.args["param1"]
param2 = self.args["param2"]
```

A use case for this might be an App that detects motion and turns on a light. If you have 3 places you want to run this, rather than hardcoding this into 3 separate Apps, you need only code a single app and instantiate it 3 times with different arguments. It might look something like this:

```
downstairs_motion_light:
  module: motion_light
  class: MotionLight
  sensor: binary_sensor.downstairs_hall
  light: light.downstairs_hall
upstairs_motion_light:
  module: motion_light
  class: MotionLight
  sensor: binary_sensor.upstairs_hall
  light: light.upstairs_hall
garage_motion_light:
  module: motion_light
  class: MotionLight
  sensor: binary_sensor.garage
  light: light.garage
```

Apps can use arbitrarily complex structures within argumens, e.g.:

```
entities:
  - entity1
  - entity2
  - entity3
```

Which can be accessed as a list in python with:

```
for entity in self.args.entities:
  do some stuff
```

Also, this opens the door to really complex parameter structures if required:

```
sensors:
  sensor1:
    type:thermometer
    warning_level: 30
    units: degrees
  sensor2:
    type:moisture
    warning_level: 100
    units: %
```

4.8 Module Dependencies

It is possible for modules to be dependant upon other modules. Some examples where this might be the case are:

- A Global module that defines constants for use in other modules
- A module that provides a service for other modules, e.g. a TTS module
- A Module that provides part of an object hierarchy to other modules

In these cases, when changes are made to one of these modules, we also want the modules that depend upon them to be reloaded. Furthermore, we also want to guarantee that they are loaded in order so that the modules dpended upon by other modules are loaded first.

AppDaemon fully supports this through the use of the dependency directive in the App configuration. Using this directice, each App identifies modules that it depends upon. Note that the dependency is at the module level, not the App level, since a change to the module will force a reload of all apps using it anyway. The dependency directive will identify the module name of the App it cares about, and AppDaemon will see to it that the dependency is loaded before the module depending on it, and that the dependent module will be reloaded if it changes.

For example, an App `Consumer`, uses another app `Sound` to play sound files. `Sound` in turn uses `Global` to store some global values. We can represent these dependencies as follows:

```
Global:
  module: global
  class: Global

Sound
  module: sound
  class: Sound
  dependencies: global # Note - module name not App name

Consumer:
```

```
  module: sound
  class: Sound
  dependencies: sound
```

It is also possible to have multiple dependencies, added as a comma separate list (no spaces)

```
Consumer:
  module: sound
  class: Sound
  dependencies: sound,global
```

AppDaemon will write errors to the log if a dependency is missing and it should also detect circular dependencies.

4.9 Callback Constraints

Callback constraints are a feature of AppDaemon that removes the need for repetition of some common coding checks. Many Apps will wish to process their callbacks only when certain conditions are met, e.g. someone is home, and it's after sunset. These kinds of conditions crop up a lot, and use of callback constraints can significantly simplify the logic required within callbacks.

Put simply, callback constraints are one or more conditions on callback execution that can be applied to an individual App. An App's callbacks will only be executed if all of the constraints are met. If a constraint is absent it will not be checked for.

For example, the presence callback constraint can be added to an App by adding a parameter to it's configuration like this:

```
some_app:
  module: some_module
  class: SomeClass
  constrain_presence: noone
```

Now, although the `initialize()` function will be called for SomeClass, and it will have a chance to register as many callbacks as it desires, none of the callbacks will execute, in this case, until everyone has left. This could be useful for an interior motion detector App for instance. There are several different types of constraints:

- input_boolean
- input_select
- presence
- time

An App can have as many or as few as are required. When more than one constraint is present, they must all evaluate to true to allow the callbacks to be called. Constraints becoming true are not an event in their own right, but if they are all true at a point in time, the next callback that would otherwise been blocked due to constraint failure will now be called. Similarly, if one of the constraints becomes false, the next callback that would otherwise have been called will be blocked.

They are described individually below.

4.9.1 input_boolean

By default, the input_boolean constraint prevents callbacks unless the specified input_boolean is set to "on". This is useful to allow certain Apps to be turned on and off from the user interface. For example:

```
some_app:
  module: some_module
  class: SomeClass
  constrain_input_boolean: input_boolean.enable_motion_detection
```

If you want to reverse the logic so the constraint is only called when the input_boolean is off, use the optional state parameter by appending ",off" to the argument, e.g.:

```
some_app:
  module: some_module
  class: SomeClass
  constrain_input_boolean: input_boolean.enable_motion_detection,off
```

4.9.2 input_select

The input_select constraint prevents callbacks unless the specified input_select is set to one or more of the nominated (comma separated) values. This is useful to allow certain Apps to be turned on and off according to some flag, e.g. a house mode flag.

```
 Single value
constrain_input_select: input_select.house_mode,Day
 or multiple values
constrain_input_select: input_select.house_mode,Day,Evening,Night
```

4.9.3 presence

The presence constraint will constrain based on presence of device trackers. It takes 3 possible values: - `noone` - only allow callback execution when no one is home - `anyone` - only allow callback execution when one or more person is home - `everyone` - only allow callback execution when everyone is home

```
constrain_presence: anyone
 or
constrain_presence: someone
 or
constrain_presence: noone
```

4.9.4 time

The time constraint consists of 2 variables, `constrain_start_time` and `constrain_end_time`. Callbacks will only be executed if the current time is between the start and end times. - If both are absent no time constraint will exist - If only start is present, end will default to 1 second before midnight - If only end is present, start will default to midnight

The times are specified in a string format with one of the following formats: - HH:MM:SS - the time in Hours Minutes and Seconds, 24 hour format. - `sunrise|sunset` [+|- HH:MM:SS]- time of the next sunrise or sunset with an optional positive or negative offset in Hours Minutes and seconds

The time based constraint system correctly interprets start and end times that span midnight.

```
 Run between 8am and 10pm
constrain_start_time: 08:00:00
constrain_end_time: 22:00:00
 Run between sunrise and sunset
```

```
constrain_start_time: sunrise
constrain_end_time: sunset
 Run between 45 minutes before sunset and 45 minutes after sunrise the next day
constrain_start_time: sunset - 00:45:00
constrain_end_time: sunrise + 00:45:00
```

4.9.5 days

The day constraint consists of as list of days for which the callbacks will fire, e.g.

```
constrain_days: mon,tue,wed
```

Callback constraints can also be applied to individual callbacks within Apps, see later for more details.

4.10 A Note on Threading

AppDaemon is multithreaded. This means that any time code within an App is executed, it is executed by one of many threads. This is generally not a particularly important consideration for this application; in general, the execution time of callbacks is expected to be far quicker than the frequency of events causing them. However, it should be noted for completeness, that it is certainly possible for different pieces of code within the App to be executed concurrently, so some care may be necessary if different callback for instance inspect and change shared variables. This is a fairly standard caveat with concurrent programming, and if you know enough to want to do this, then you should know enough to put appropriate safeguards in place. For the average user however this shouldn't be an issue. If there are sufficient use cases to warrant it, I will consider adding locking to the function invocations to make the entire infrastructure threadsafe, but I am not convinced that it is necessary.

An additional caveat of a threaded worker pool environment is that it is the expectation that none of the callbacks tie threads up for a significant amount of time. To do so would eventually lead to thread exhaustion, which would make the system run behind events. No events would be lost as they would be queued, but callbacks would be delayed which is a bad thing.

Given the above, NEVER use Python's `time.sleep()` if you want to perform an operation some time in the future, as this will tie up a thread for the period of the sleep. Instead use the scheduler's `run_in()` function which will allow you to delay without blocking any threads.

4.11 State Operations

4.11.1 A note on Home Assistant State

State within Home Assistant is stored as a collection of dictionaries, one for each entity. Each entity's dictionary will have some common fields and a number of entity type specific fields The state for an entity will always have the attributes:

- `last_updated`
- `last_changed`
- `state`

Any other attributes such as brightness for a lamp will only be present if the entity supports them, and will be stored in a sub-dictionary called `attributes`. When specifying these optional attributes in the `get_state()` call, no

special distinction is required between the main attributes and the optional ones - `get_state()` will figure it out for you.

Also bear in mind that some attributes such as brightness for a light, will not be present when the light is off.

In most cases, the attribute `state` has the most important value in it, e.g. for a light or switch this will be `on` or `off`, for a sensor it will be the value of that sensor. Many of the AppDaemon API calls and callbacks will implicitly return the value of state unless told to do otherwise.

Although the use of `get_state()` (below) is still supported, as of AppDaemon 2.0.9 it is easier to access HASS state directly as an attribute of the App itself, under the `entities` attribute.

For instance, to access the state of a binary sensor, you could use:

```
sensor_state = self.entities.binary_sensor.downstairs_sensor.state
```

Similarly, accessing any of the entity attributes is also possible:

```
name = self.entities.binary_sensor.downstairs_sensor.attributes.friendly_name
```

4.11.2 About Callbacks

A large proportion of home automation revolves around waiting for something to happen and then reacting to it; a light level drops, the sun rises, a door opens etc. Home Assistant keeps track of every state change that occurs within the system and streams that information to AppDaemon almost immediately.

An individual App however usually doesn't care about the majority of state changes going on in the system; Apps usually care about something very specific, like a specific sensor or light. Apps need a way to be notified when a state change happens that they care about, and be able to ignore the rest. They do this through registering callbacks. A callback allows the App to describe exactly what it is interested in, and tells AppDaemon to make a call into its code in a specific place to be able to react to it - this is a very familiar concept to anyone familiar with event-based programming.

There are 3 types of callbacks within AppDaemon:

- State Callbacks - react to a change in state
- Scheduler Callbacks - react to a specific time or interval
- Event Callbacks - react to specific Home Assistant and Appdaemon events.

All callbacks allow the user to specify additional parameters to be handed to the callback via the standard Python `**kwargs` mechanism for greater flexibility, these additional arguments are handed to the callback as a standard Python dictionary,

4.11.3 About Registering Callbacks

Each of the various types of callback have their own function or functions for registering the callback:

- `listen_state()` for state callbacks
- Various scheduler calls such as `run_once()` for scheduler callbacks
- `listen_event()` for event callbacks.

Each type of callback shares a number of common mechanisms that increase flexibility.

Callback Level Constraints

When registering a callback, you can add constraints identical to the Application level constraints described earlier. The difference is that a constraint applied to an individual callback only affects that callback and no other. The constraints are applied by adding Python keyword-value style arguments after the positional arguments. The parameters themselves are named identically to the previously described constraints and have identical functionality. For instance, adding:

```
constrain_presence="everyone"
```

to a callback registration will ensure that the callback is only run if the callback conditions are met and in addition everyone is present although any other callbacks might run whenever their event fires if they have no constraints.

For example:

```
self.listen_state(self.motion, "binary_sensor.drive", constrain_presence="everyone")
```

User Arguments

Any callback has the ability to allow the App creator to pass through arbitrary keyword arguments that will be presented to the callback when it is run. The arguments are added after the positional parameters just like the constraints. The only restriction is that they cannot be the same as any constraint name for obvious reasons. For example, to pass the parameter `arg1 = "home assistant"` through to a callback you would register a callback as follows:

```
self.listen_state(self.motion, "binary_sensor.drive", arg1="home assistant")
```

Then in the callback it is presented back to the function as a dictionary and you could use it as follows:

```
def motion(self, entity, attribute, old, new, kwargs):
    self.log("Arg1 is {}".format(kwargs["arg1"]))
```

4.11.4 State Callbacks

AppDaemons's state callbacks allow an App to listen to a wide variety of events, from every state change in the system, right down to a change of a single attribute of a particular entity. Setting up a callback is done using a single API call `listen_state()` which takes various arguments to allow it to do all of the above. Apps can register as many or as few callbacks as they want.

4.11.5 About State Callback Functions

When calling back into the App, the App must provide a class function with a known signature for AppDaemon to call. The callback will provide various information to the function to enable the function to respond appropriately. For state callbacks, a class defined callback function should look like this:

```
def my_callback(self, entity, attribute, old, new, kwargs):
  <do some useful work here>
```

You can call the function whatever you like - you will reference it in the `listen_state()` call, and you can create as many callback functions as you need.

The parameters have the following meanings:

self

A standard Python object reference.

entity

Name of the entity the callback was requested for or `None`.

attribute

Name of the attribute the callback was requested for or `None`.

old

The value of the state before the state change.

new

The value of the state after the state change.

`old` and `new` will have varying types depending on the type of callback.

**kwargs

A dictionary containing any constraints and/or additional user specific keyword arguments supplied to the `listen_state()` call.

4.12 Publishing State from an App

Using AppDaemon it is possible to explicitly publish state from an App. The published state can contain whatever you want, and is treated exactly like any other HA state, e.g. to the rest of AppDaemon, and the dashboard it looks like an entity. This means that you can listen for state changes in other apps and also publish arbitary state to the dashboard via use of specific entity IDs. To publish state, you will use `set_app_state()`. State can be retrieved and listened for with the usual AppDaemon calls.

4.13 The Scheduler

AppDaemon contains a powerful scheduler that is able to run with 1 second resolution to fire off specific events at set times, or after set delays, or even relative to sunrise and sunset. In general, events should be fired less than a second after specified but under certain circumstances there may be short additional delays.

4.13.1 About Schedule Callbacks

As with State Change callbacks, Scheduler Callbacks expect to call into functions with a known and specific signature and a class defined Scheduler callback function should look like this:

```
def my_callback(self, kwargs):
  <do some useful work here>
```

You can call the function whatever you like; you will reference it in the Scheduler call, and you can create as many callback functions as you need.

The parameters have the following meanings:

self

A standard Python object reference

**kwargs

A dictionary containing Zero or more keyword arguments to be supplied to the callback.

4.13.2 Creation of Scheduler Callbacks

Scheduler callbacks are created through use of a number of convenience functions which can be used to suit the situation.

4.13.3 Scheduler Randomization

All of the scheduler calls above support 2 additional optional arguments, `random_start` and `random_end`. Using these arguments it is possible to randomize the firing of callbacks to the degree desired by setting the appropriate number of seconds with the parameters.

- `random_start` - start of range of the random time
- `random_end` - end of range of the random time

`random_start` must always be numerically lower than `random_end`, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event. The event would be a an absolute or relative time or sunrise/sunset depending on whcih scheduler call you use and these values affect the base time by the spcified amount. If not specified, they will default to `0`.

For example:

```
 Run a callback in 2 minutes minus a random number of seconds between 0 and 60, e.g.␣
↪run between 60 and 120 seconds from now
self.handle = self.run_in(callback, 120, random_start = -60, **kwargs)
 Run a callback in 2 minutes plus a random number of seconds between 0 and 60, e.g.␣
↪run between 120 and 180 seconds from now
self.handle = self.run_in(callback, 120, random_end = 60, **kwargs)
 Run a callback in 2 minutes plus or minus a random number of seconds between 0 and␣
↪60, e.g. run between 60 and 180 seconds from now
self.handle = self.run_in(callback, 120, random_start = -60, random_end = 60,␣
↪**kwargs)
```

4.14 Sunrise and Sunset

AppDaemon has a number of features to allow easy tracking of sunrise and sunset as well as a couple of scheduler functions. Note that the scheduler functions also support the randomization parameters described above, but they cannot be used in conjunction with the `offset` parameter‘.

4.15 Calling Services

4.15.1 About Services

Services within Home Assistant are how changes are made to the system and its devices. Services can be used to turn lights on and off, set thermostats and a whole number of other things. Home Assistant supplies a single interface to all these disparate services that take arbitrary parameters. AppDaemon provides the `call_service()` function to call into Home Assistant and run a service. In addition, it also provides convenience functions for some of the more common services making calling them a little easier.

4.16 Events

4.16.1 About Events

Events are a fundamental part of how Home Assistant works under the covers. HA has an event bus that all components can read and write to, enabling components to inform other components when important events take place. We have already seen how state changes can be propagated to AppDaemon - a state change however is merely an example of an event within Home Assistant. There are several other event types, among them are:

- `homeassistant_start`
- `homeassistant_stop`
- `state_changed`
- `service_registered`
- `call_service`
- `service_executed`
- `platform_discovered`
- `component_loaded`

Using AppDaemon, it is possible to subscribe to specific events as well as fire off events.

In addition to the Home Assistant supplied events, AppDaemon adds 2 more events. These are internal to AppDaemon and are not visible on the Home Assistant bus:

- `appd_started` - fired once when AppDaemon is first started and after Apps are initialized
- `ha_started` - fired every time AppDaemon detects a Home Assistant restart
- `ha_disconnectd` - fired once every time AppDaemon loses its connection with HASS

4.16.2 About Event Callbacks

As with State Change and Scheduler callbacks, Event Callbacks expect to call into functions with a known and specific signature and a class defined Scheduler callback function should look like this:

```
def my_callback(self, event_name, data, kwargs):
  <do some useful work here>
```

You can call the function whatever you like - you will reference it in the Scheduler call, and you can create as many callback functions as you need.

The parameters have the following meanings:

self

A standard Python object reference.

event_name

Name of the event that was called, e.g. `call_service`.

data

Any data that the system supplied with the event as a dict.

kwargs

A dictionary containing Zero or more user keyword arguments to be supplied to the callback.

4.16.3 listen_event()

Listen event sets up a callback for a specific event, or any event.

Synopsis

```
handle = listen_event(function, event = None, **kwargs):
```

Returns

A handle that can be used to cancel the callback.

Parameters

function

The function to be called when the event is fired.

event

Name of the event to subscribe to. Can be a standard Home Assistant event such as `service_registered` or an arbitrary custom event such as `"MODE_CHANGE"`. If no event is specified, `listen_event()` will subscribe to all events.

**kwargs (optional)

One or more keyword value pairs representing App specific parameters to supply to the callback. If the keywords match values within the event data, they will act as filters, meaning that if they don't match the values, the callback will not fire.

As an example of this, a Minimote controller when activated will generate an event called `zwave.scene_activated`, along with 2 pieces of data that are specific to the event - `entity_id` and `scene`. If you include keyword values for either of those, the values supplied to the 'listen_event()1 call must match the values in the event or it will not fire. If the keywords do not match any of the data in the event they are simply ignored.

Filtering will work with any event type, but it will be necessary to figure out the data associated with the event to understand what values can be filtered on. This can be achieved by examining Home Assistant's logfiles when the event fires.

Examples

```
self.listen_event(self.mode_event, "MODE_CHANGE")
 Listen for a minimote event activating scene 3:
self.listen_event(self.generic_event, "zwave.scene_activated", scene_id = 3)
 Listen for a minimote event activating scene 3 from a specific minimote:
self.listen_event(self.generic_event, "zwave.scene_activated", entity_id = "minimote_
↪31", scene_id = 3)
```

4.16.4 Use of Events for Signalling between Home Assistant and AppDaemon

Home Assistant allows for the creation of custom events and existing components can send and receive them. This provides a useful mechanism for signaling back and forth between Home Assistant and AppDaemon. For instance, if you would like to create a UI Element to fire off some code in Home Assistant, all that is necessary is to create a script to fire a custom event, then subscribe to that event in AppDaemon. The script would look something like this:

```
alias: Day
sequence:
- event: MODE_CHANGE
  event_data:
    mode: Day
```

The custom event `MODE_CHANGE` would be subscribed to with:

```
self.listen_event(self.mode_event, "MODE_CHANGE")
```

Home Assistant can send these events in a variety of other places - within automations, and also directly from Alexa intents. Home Assistant can also listen for custom events with it's automation component. This can be used to signal from AppDaemon code back to home assistant. Here is a sample automation:

```
automation:
  trigger:
    platform: event
    event_type: MODE_CHANGE
    ...
    ...
```

This can be triggered with a call to AppDaemon's fire_event() as follows:

```
self.fire_event("MODE_CHANGE", mode = "Day")
```

4.16.5 Use of Events for Interacting with HADashboard

HADashboard listens for certain events. An event type of "hadashboard" will trigger certain actions such as page navigation. For more information see the ' Dashboard configuration pages <DASHBOARD.html>'__

AppDaemon provides convenience funtions to assist with this.

4.17 Presence

Presence in Home Assistant is tracked using Device Trackers. The state of all device trackers can be found using the `get_state()` call, however AppDaemon provides several convenience functions to make this easier.

4.17.1 Writing to Logfiles

AppDaemon uses 2 separate logs - the general log and the error log. An AppDaemon App can write to either of these using the supplied convenience methods `log()` and `error()`, which are provided as part of parent `AppDaemon` class, and the call will automatically pre-pend the name of the App making the call. The `-D` option of AppDaemon can be used to specify what level of logging is required and the logger objects will work as expected.

ApDaemon loggin also allows you to use placeholders for the module, fucntion and line number. If you include the following in the test of your message:

```
__function__
__module__
__line__
```

They will automatically be expanded to the appropriate values in the log message.

4.18 Getting Information in Apps and Sharing information between Apps

Sharing information between different Apps is very simple if required. Each app gets access to a global dictionary stored in a class attribute called `self.global_vars`. Any App can add or read any key as required. This operation is not however threadsafe so some care is needed.

In addition, Apps have access to the entire configuration if required, meaning they can access AppDaemon configuration items as well as parameters from other Apps. To use this, there is a class attribute called `self.config`. It contains a `ConfigParser` object, which is similar in operation to a `Dictionary`. To access any apps parameters, simply reference the ConfigParser object using the Apps name (form the config file) as the first key, and the parameter required as the second, for instance:

```
other_apps_arg = self.config["some_app"]["some_parameter"].
```

To get AppDaemon's config parameters, use the key "AppDaemon", e.g.:

```
app_timezone = self.config["AppDaemon"]["time_zone"]
```

AppDaemon also exposes configuration from Home Assistant such as the Latitude and Longitude configured in HA. All of the information available from the Home Assistant `/api/config` endpoint is available in the `self.ha_config` dictionary. E.g.:

```
self.log("My current position is {}(Lat), {}(Long)".format(self.ha_config["latitude"],
↪ self.ha_config["longitude"]))
```

And finally, it is also possible to use the AppDaemon as a global area for sharing parameters across Apps. Simply add the required parameters to the AppDaemon section of your config:

```
AppDaemon:
ha_url: <some url>
ha_key: <some key>
...
global_var: hello world
```

Then access it as follows:

```
my_global_var = conf.config["AppDaemon"]["global_var"]
```

4.19 Development Workflow

Developing Apps is intended to be fairly simple but is an exercise in programming like any other kind of Python programming. As such, it is expected that apps will contain syntax errors and will generate exceptions during the development process. AppDaemon makes it very easy to iterate through the development process as it will automatically reload code that has changed and also will reload code if any of the parameters in the configuration file change as well.

The recommended workflow for development is as follows:

- Open a window and tail the `appdaemon.log` file
- Open a second window and tail the `error.log` file
- Open a third window or the editor of your choice for editing the App

With this setup, you will see that every time you write the file, AppDaemon will log the fact and let you know it has reloaded the App in the `appdaemon.log` file.

If there is an error in the compilation or a runtime error, this will be directed to the `error.log` file to enable you to see the error and correct it. When an error occurs, there will also be a warning message in `appdaemon.log` to tell you to check the error log.

4.20 Time Travel

OK, time travel sadly isn't really possible but it can be very useful when testing Apps. For instance, imagine you have an App that turns a light on every day at sunset. It might be nice to test it without waiting for Sunset - and with AppDaemon's "Time Travel" features you can.

4.20.1 Choosing a Start Time

Internally, AppDaemon keeps track of it's own time relative to when it was started. This make is possible to start AppDaemon with a different start time and date to the current time. For instance to test that sunset App, start AppDaemon at a time just before sunset and see if it works as expected. To do this, simply use the "-s" argument on AppDaemon's command line. e,g,:

```
$ appdaemon -s "2016-06-06 19:16:00"
2016-09-06 17:16:00 INFO AppDaemon Version 1.3.2 starting
2016-09-06 17:16:00 INFO Got initial state
2016-09-06 17:16:00 INFO Loading Module: /export/hass/appdaemon_test/conf/test_apps/
↪sunset.py
...
```

Note the timestamps in the log - AppDaemon believes it is now just before sunset and will process any callbacks appropriately.

4.20.2 Speeding things up

Some Apps need to run for periods of a day or two for you to test all aspects. This can be time consuming, but Time Travel can also help here in two ways. The first is by speeding up time. To do this, simply use the `-t` option on the command line. This specifies the amount of time a second lasts while time travelling. The default of course is 1 second, but if you change it to `0.1` for instance, AppDaemon will work 10x faster. If you set it to `0`, AppDaemon will work as fast as possible and, depending in your hardware, may be able to get through an entire day in a matter of minutes. Bear in mind however, due to the threaded nature of AppDaemon, when you are running with `-t 0` you may see actual events firing a little later than expected as the rest of the system tries to keep up with the timer. To set the tick time, start AppDaemon as follows:

```
$ appdaemon -t 0.1
```

AppDaemon also has an interval flag - think of this as a second multiplier. If the flag is set to 3600 for instance, each tick of the scheduler will jump the time forward by an hour. This is good for covering vast amounts of time quickly but event firing accuracy will suffer as a result. For example:

```
$ appdaemon -i 3600
```

4.20.3 Automatically stopping

AppDaemon can be set to terminate automatically at a specific time. This can be useful if you want to repeatedly rerun a test, for example to test that random values are behaving as expected. Simply specify the end time with the `-e` flag as follows:

```
$ appdaemon -e "2016-06-06 10:10:00"
2016-09-06 17:16:00 INFO AppDaemon Version 1.3.2 starting
2016-09-06 17:16:00 INFO Got initial state
2016-09-06 17:16:00 INFO Loading Module: /export/hass/appdaemon_test/conf/test_apps/
↪sunset.py
..,
```

The `-e` flag is most useful when used in conjuntion with the `-s` flag and optionally the `-t` flag. For example, to run from just before sunset, for an hour, as fast as possible:

```
$ appdaemon -s "2016-06-06 19:16:00" -e "2016-06-06 20:16:00" -t 0
```

4.20.4 A Note On Times

Some Apps you write may depend on checking times of events relative to the current time. If you are time travelling this will not work if you use standard python library calls to get the current time and date etc. For this reason, always

use the AppDamon supplied `time()`, `date()` and `datetime()` calls, documented earlier. These calls will consult with AppDaemon's internal time rather than the actual time and give you the correct values.

4.20.5 Other Functions

AppDaemon allows some introspection on its stored schedule and callbacks which may be useful for some applications. The functions:

- get_scheduler_entries()
- get_callback_entries()

Return the internal data structures, but do not allow them to be modified directly. Their format may change.

4.20.6 About HASS Disconnections

When AppDaemon is unable to connect initially with Home Assistant, it will hold all Apps in statsis until it initially connects, nothing else will happen and no initialization routines will be called. If AppDaemon has been running connected to Home Assitant for a while and the connection is unexpectedly lost, the following will occur:

- When HASS first goes down or becomes disconnected, an event called `ha_disconnected` will fire
- While disconnected from HASS, Apps will continue to run
- Schedules will continue to be honored
- Any operation reading locally cached state will succeed
- Any operation requiring a call to HASS will log a warning and return without attempting to contact hass
- Changes to Apps will not force a reload until HASS is reconnected

When a connection to HASS is reestablished, all Apps will be restarted and their `initialize()` routines will be called.

4.21 RESTFul API Support

AppDaemon supports a simple RESTFul API to enable arbitary HTTP connections to pass data to Apps and trigger actions. API Calls must use a content type of `application/json`, and the response will be JSON encoded. The RESTFul API is disabled by default, but is enabled by adding an `ad_port` directive to the AppDaemon section of the configuration file. The API can run http or https if desired, separately from the dashboard.

To call into a specific App, construct a URL, use the regular HADashboard URL, and append `/api/appdaemon`, then add the name of the endpoint as registered by the app on the end, for example:

```
http://192.168.1.20:5050/api/appdaemon/hello_endpoint
```

This URL will call into an App that registered an endpoint named `hello_endpoint`.

Within the app, a call must be made to `register_endpoint()` to tell AppDaemon that the app is expecting calls on that endpoint. When registering an endpoint, the App supplies a function to be called when a request comes in to that endpoint and an optional name for the endpoint. If not specified, the name will default to the name of the App as specified in the configuration file.

Apps can have as many endpoints as required, however the names must be unique across all of the Apps in an AppDaemon instance.

It is also possible to remove endpoints with the `unregister_endpoint()` call, making the endpoints truly dynamic and under the control of the App.

Here is an example of an App using the API:

```
import appdaemon.appapi as appapi

class API(appapi.AppDaemon):

    def initialize(self):
        self.register_endpoint(my_callback, test_endpoint)

    def my_callback(self, data):

        self.log(data)

        response = {"message": "Hello World"}

        return response, 200
```

The response must be a python structure that can be mapped to JSON, or can be blank, in which case specify `""` for the response. You should also return an HTML status code, that will be reported back to the caller, `200` should be used for an OK response.

As well as any user specified code, the API can return the following codes:

- 400 - JSON Decode Error
- 401 - Unauthorized
- 404 - App not found

Below is an example of using curl to call into the App shown above:

```
hass@Pegasus:~$ curl -i -X POST -H "Content-Type: application/json" http://192.168.1.
↪20:5050/api/appdaemon/test_endpoint -d '{"type": "Hello World Test"}'
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 26
Date: Sun, 06 Aug 2017 16:38:14 GMT
Server: Python/3.5 aiohttp/2.2.3

{"message": "Hello World"}hass@Pegasus:~$
```

4.22 API Security

If you have added a key to the AppDaemon config, AppDaemon will expect to find a header called "x-ad-access" in the request with a value equal to the configured key. A security key is added for the API with the `api_key` directive described in the Installation Documentation

If these conditions are not met, the call will fail with a return code of `401 Not Authorized`. Here is a succesful curl example:

```
hass@Pegasus:~$ curl -i -X POST -H "x-ad-access: fred" -H "Content-Type: application/
↪json" http://192.168.1.20:5050/api/appdaemon/api -d '{"type": "Hello World
 Test"}'
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 26
Date: Sun, 06 Aug 2017 17:30:50 GMT
Server: Python/3.5 aiohttp/2.2.3

{"message": "Hello World"}hass@Pegasus:~$
```

And an example of a missing key:

```
hass@Pegasus:~$ curl -i -X POST -H "Content-Type: application/json" http://192.168.1.
↪20:5050/api/appdaemon/api Test"}'ype": "Hello World
HTTP/1.1 401 Unauthorized
Content-Length: 112
Content-Type: text/plain; charset=utf-8
Date: Sun, 06 Aug 2017 17:30:43 GMT
Server: Python/3.5 aiohttp/2.2.3

<html><head><title>401 Unauthorized</title></head><body><h1>401 Unauthorized</h1>
↪Error in API Call</body></html>hass@Pegasus:~$
```

4.23 Alexa Support

AppDaemon is able to use the API support to accept calls from Alexa. Amazon Alexa calls can be directed to AppDaemon and arrive as JSON encoded requests. AppDaemon provides several helper functions to assist in understanding the request and responding appropriately. Since Alexa only allows one URL per skill, the mapping will be 1:1 between skills and Apps. When constructing the URL in the Alexa Intent, make sure it points to the correct endpoint for the App you are using for Alexa.

In addition, if you are using API security keys (recommended) you will need to append it to the end of the url as follows:

```
http://<some.host.com>/api/appdaemon/alexa?api_password=<password>
```

For more information about configuring Alexa Intents, see the Home Assistant Alexa Documentation

When configuring Alexa support for AppDaemon some care is needed. If as most people are, you are using SSL to access Home Assistant, there is contention for use of the SSL port (443) since Alexa does not allow you to change this. This means that if you want to use AppDaemon with SSL, you will not be able to use Home Assistant remotely over SSL. The way around this is to use NGINX to remap the specific AppDamon API URL to a different port, by adding something like this to the config:

```
  location /api/appdaemon/ {
  allow all;
  proxy_pass http://localhost:5000;
  proxy_set_header Host $host;
  proxy_redirect http:// http://;
}
```

Here we see the default port being remapped to port 5000 which is where AppDamon is listening in my setup.

Since each individual Skill has it's own URL it is possible to have different skills for Home Assitant and AppDaemon.

4.24 Putting it together in an App

The Alexa App is basically just a standard API App that uses Alexa helper functions to understand the incoming request and format a response to be sent back to Amazon, to describe the spoken resonse and card for Alexa.

Here is a sample Alexa App that can be extended for whatever intents you want to configure.

```
import appdaemon.appapi as appapi
import random
import globals

class Alexa(appapi.AppDaemon):

    def initialize(self):
        pass

    def api_call(self, data):
        intent = self.get_alexa_intent(data)

        if intent is None:
            self.log("Alexa error encountered: {}".format(self.get_alexa_error(data)))
            return "", 201

        intents = {
            "StatusIntent": self.StatusIntent,
            "LocateIntent": self.LocateIntent,
        }

        if intent in intents:
            speech, card, title = intents[intent](data)
            response = self.format_alexa_response(speech = speech, card = card, title␣
↪= title)
            self.log("Recieved Alexa request: {}, answering: {}".format(intent,␣
↪speech))
        else:
            response = self.format_alexa_response(speech = "I'm sorry, the {} does␣
↪not exist within AppDaemon".format(intent))

        return response, 200

    def StatusIntent(self, data):
        response = self.HouseStatus()
        return response, response, "House Status"

    def LocateIntent(self, data):
        user = self.get_alexa_slot_value(data, "User")

        if user is not None:
            if user.lower() == "jack":
                response = self.Jack()
            elif user.lower() == "andrew":
                response = self.Andrew()
            elif user.lower() == "wendy":
                response = self.Wendy()
            elif user.lower() == "brett":
                response = "I have no idea where Brett is, he never tells me anything"
            else:
                response = "I'm sorry, I don't know who {} is".format(user)
```

```
        else:
            response = "I'm sorry, I don't know who that is"

        return response, response, "Where is {}?".format(user)

    def HouseStatus(self):

        status = "The downstairs temperature is {} degrees farenheit,".format(self.
↪entities.sensor.downstairs_thermostat_temperature.state)
        status += "The upstairs temperature is {} degrees farenheit,".format(self.
↪entities.sensor.upstairs_thermostat_temperature.state)
        status += "The outside temperature is {} degrees farenheit,".format(self.
↪entities.sensor.side_temp_corrected.state)
        status += self.Wendy()
        status += self.Andrew()
        status += self.Jack()

        return status

    def Wendy(self):
        location = self.get_state(globals.wendy_tracker)
        if location == "home":
            status = "Wendy is home,"
        else:
            status = "Wendy is away,"

        return status

    def Andrew(self):
        location = self.get_state(globals.andrew_tracker)
        if location == "home":
            status = "Andrew is home,"
        else:
            status = "Andrew is away,"

        return status

    def Jack(self):
        responses = [
            "Jack is asleep on his chair",
            "Jack just went out bowling with his kitty friends",
            "Jack is in the hall cupboard",
            "Jack is on the back of the den sofa",
            "Jack is on the bed",
            "Jack just stole a spot on daddy's chair",
            "Jack is in the kitchen looking out of the window",
            "Jack is looking out of the front door",
            "Jack is on the windowsill behind the bed",
            "Jack is out checking on his clown suit",
            "Jack is eating his treats",
            "Jack just went out for a walk in the neigbourhood",
            "Jack is by his bowl waiting for treats"
        ]

        return random.choice(responses)
```

4.25 Google API.AI

Similarly, Google's API.AI for Google home is supported - here is the Google version of the same App.To set up Api.ai with your google home refer to the apiai component in home-assistant. Once it is setup you can use the appdaemon API as the webhook.

import appdaemon.appapi as appapi import random import globals

class Apiai(appapi.AppDaemon):

def initialize(self): pass

def api_call(self, data): intent = self.get_apiai_intent(data)

if intent is None: self.log("Apiai error encountered: Result is empty") return "", 201

intents = { "StatusIntent": self.StatusIntent, "LocateIntent": self.LocateIntent,

}

if intent in intents: speech = intents[intent](data) response = self.format_apiai_response(speech) self.log("Recieved Apai request: {}, answering: {}".format(intent, speech))

else: response = self.format_apaiai_response(speech = "I'm sorry, the {} does not exist within AppDaemon".format(intent))

return response, 200

def StatusIntent(self, data): response = self.HouseStatus() return response

def LocateIntent(self, data): user = self.get_apiai_slot_value(data, "User")

if user is not None:

if user.lower() == "jack": response = self.Jack()

elif user.lower() == "andrew": response = self.Andrew()

elif user.lower() == "wendy": response = self.Wendy()

elif user.lower() == "brett": response = "I have no idea where Brett is, he never tells me anything"

else: response = "I'm sorry, I don't know who {} is".format(user)

else: response = "I'm sorry, I don't know who that is"

return response

def HouseStatus(self):

status = "The downstairs temperature is {} degrees farenheit,".format(self.entities.sensor.downstairs_thermostat_temperature.state) status += "The upstairs temperature is {} degrees farenheit,".format(self.entities.sensor.upstairs_thermostat_temperature.state) status += "The outside temperature is {} degrees farenheit,".format(self.entities.sensor.side_temp_corrected.state) status += self.Wendy() status += self.Andrew() status += self.Jack()

return status

def Wendy(self): location = self.get_state(globals.wendy_tracker) if location == "home":

status = "Wendy is home,"

else: status = "Wendy is away,"

return status

def Andrew(self): location = self.get_state(globals.andrew_tracker) if location == "home":

status = "Andrew is home,"

else: status = "Andrew is away,"

return status

def Jack(self):

responses = ["Jack is asleep on his chair", "Jack just went out bowling with his kitty friends", "Jack is in the hall cupboard", "Jack is on the back of the den sofa", "Jack is on the bed", "Jack just stole a spot on daddy's chair", "Jack is in the kitchen looking out of the window", "Jack is looking out of the front door", "Jack is on the windowsill behind the bed", "Jack is out checking on his clown suit", "Jack is eating his treats", "Jack just went out for a walk in the neigbourhood", "Jack is by his bowl waiting for treats"

]

return random.choice(responses)

CHAPTER 5

AppDaemon API Reference

5.1 State Operations

5.1.1 get_state()

Synopsis

```
get_state(entity = None, attribute: None)
```

`get_state()` is used to query the state of any component within Home Assistant. State updates are continuously tracked so this call runs locally and does not require AppDaemon to call back to Home Assistant and as such is very efficient.

Returns

`get_state()` returns a `dictionary` or single value, the structure of which varies according to the parameters used. If an entity or attribute does not exist, `get_state()` will return `None`.

Parameters

All parameters are optional, and if `get_state()` is called with no parameters it will return the entire state of Home Assistant at that given time. This will consist of a dictionary with a key for each entity. Under that key will be the standard entity state information.

entity

This is the name of an entity or device type. If just a device type is provided, e.g. `light` or `binary_sensor`, `get_state()` will return a dictionary of all devices of that type, indexed by the entity_id, containing all the state for each entity.

If a fully qualified `entity_id` is provided, `get_state()` will return the state attribute for that entity, e.g. `on` or `off` for a light.

attribute

Name of an attribute within the entity state object. If this parameter is specified in addition to a fully qualified `entity_id`, a single value representing the attribute will be returned, or `None` if it is not present.

The value `all` for attribute has special significance and will return the entire state dictionary for the specified entity rather than an individual attribute value.

Examples

```
 Return state for the entire system
state = self.get_state()

 Return state for all switches in the system
state = self.get_state("switch")

 Return the state attribute for light.office_1
state = self.get_state("light.office_1")

 Return the brightness attribute for light.office_1
state = self.get_state("light.office_1", "brightness")

 Return the entire state for light.office_1
state = self.get_state("light.office_1", "all")
```

5.1.2 set_state()

`set_state()` will make a call back to Home Assistant and make changes to the internal state of Home Assistant. This is not something that you would usually want to do and the applications are limited however the call is included for completeness. Note that for instance, setting the state of a light to `on` won't actually switch the device on, it will merely change the state of the device in Home Assistant so that it no longer reflects reality. In most cases, the state will be corrected the next time Home Assistant polls the device or someone causes a state change manually. To effect actual changes of devices use one of the service call functions.

One possible use case for `set_state()` is for testing. If for instance you are writing an App to turn on a light when it gets dark according to a luminance sensor, you can use `set_state()` to temporarily change the light level reported by the sensor to test your program. However this is also possible using the developer tools.

At the time of writing, it appears that no checking is done as to whether or not the entity exists, so it is possible to add entirely new entries to Home Assistant's state with this call.

Synopsis

```
set_state(entity_id, **kwargs)
```

Returns

`set_state()` returns a dictionary representing the state of the device after the call has completed.

Parameters

entity_id

Entity id for which the state is to be set, e.g. `light.office_1`.

values

A list of keyword values to be changed or added to the entities state. e.g. `state = "off"`. Note that any optional attributes such as colors for bulbs etc, need to reside in a dictionary called `attributes`; see the example.

Examples

```
status = self.set_state("light.office_1", state = "on", attributes = {"color_name":
↪"red"})
```

5.1.3 listen_state()

`listen_state()` allows the user to register a callback for a wide variety of state changes.

Synopsis

```
handle = listen_state(callback, entity = None, **kwargs)
```

Returns

A unique identifier that can be used to cancel the callback if required. Since variables created within object methods are local to the function they are created in, and in all likelihood the cancellation will be invoked later in a different function, it is recommended that handles are stored in the object namespace, e.g. `self.handle`.

Parameters

All parameters except `callback` are optional, and if `listen_state()` is called with no additional parameters it will subscribe to any state change within Home Assistant.

callback

Function to be invoked when the requested state change occurs. It must conform to the standard State Callback format documented above.

entity

This is the name of an entity or device type. If just a device type is provided, e.g. `light` or `binary_sensor`, `listen_state()` will subscribe to state changes of all devices of that type. If a fully qualified `entity_id` is provided, `listen_state()` will listen for state changes for just that entity.

When called, AppDaemon will supply the callback function, in old and new, with the state attribute for that entity, e.g. `on` or `off` for a light.

attribute (optional)

Name of an attribute within the entity state object. If this parameter is specified in addition to a fully qualified `entity_id`, `listen_state()` will subscribe to changes for just that attribute within that specific entity. The new and old parameters in the callback function will be provided with a single value representing the attribute.

The value `all` for attribute has special significance and will listen for any state change within the specified entity, and supply the callback functions with the entire state dictionary for the specified entity rather than an individual attribute value.

new = (optional)

If `new` is supplied as a parameter, callbacks will only be made if the state of the selected attribute (usually `state`) in the new state match the value of `new`.

old = (optional)

If `old` is supplied as a parameter, callbacks will only be made if the state of the selected attribute (usually `state`) in the old state match the value of `old`.

Note: `old` and `new` can be used singly or together.

duration = (optional)

If duration is supplied as a parameter, the callback will not fire unless the state listened for is maintained for that number of seconds. This makes the most sense if a specific attribute is specified (or the default of `state` is used), and in conjunction with the `old` or `new` parameters, or both. When the callback is called, it is supplied with the values of `entity`, `attr`, `old` and `new` that were current at the time the actual event occured, since the assumption is that none of them have changed in the intervening period.

immediate = (optional)

True or False

Quick check enables the countdown for a `delay` parameter to start at the time the callback is registered, rather than requiring one or more state changes. This can be useful if for instance you want the duration to be triggered immediately if a light is already on.

If `immediate` is in use, and `new` and `duration` are both set, AppDaemon will check if the entity is already set to the new state and if so it will start the clock immediately. In this case, old will be ignored and when the timer triggers, it's state will be set to None. If new or entity are not set, `immediate` will be ignored.

**kwargs

Zero or more keyword arguments that will be supplied to the callback when it is called.

Examples

```
# Listen for any state change and return the state attribute
self.handle = self.listen_state(self.my_callback)

# Listen for any state change involving a light and return the state attribute
self.handle = self.listen_state(self.my_callback, "light")

# Listen for a state change involving light.office1 and return the state attribute
self.handle = self.listen_state(self.my_callback, "light.office_1")

# Listen for a state change involving light.office1 and return the entire state as a␣
↪dict
self.handle = self.listen_state(self.my_callback, "light.office_1", attribute = "all")

# Listen for a state change involving the brightness attribute of light.office1
self.handle = self.listen_state(self.my_callback, "light.office_1", attribute =
↪"brightness")

# Listen for a state change involving light.office1 turning on and return the state␣
↪attribute
self.handle = self.listen_state(self.my_callback, "light.office_1", new = "on")

# Listen for a state change involving light.office1 changing from brightness 100 to␣
↪200 and return the state attribute
self.handle = self.listen_state(self.my_callback, "light.office_1", old = "100", new␣
↪= "200")

# Listen for a state change involving light.office1 changing to state on and␣
↪remaining on for a minute
self.handle = self.listen_state(self.my_callback, "light.office_1", new = "on",␣
↪duration = 60)

# Listen for a state change involving light.office1 changing to state on and␣
↪remaining on for a minute
# Trigger the delay immediately if the light is already on
self.handle = self.listen_state(self.my_callback, "light.office_1", new = "on",␣
↪duration = 60, immediate = True)
```

5.1.4 cancel_listen_state()

Cancel a `listen_state()` callback. This will mean that the App will no longer be notified for the specific state change that has been cancelled. Other state changes will continue to be monitored.

Synopsis

```
cancel_listen_state(handle)
```

Returns

Nothing

Parameters

handle

The handle returned when the `listen_state()` call was made.

Examples

```
self.cancel_listen_state(self.office_light_handle)
```

5.1.5 info_listen_state()

Get information on state a callback from it's handle.

Synopsis

```
entity, attribute, kwargs = self.info_listen_state(self.handle)
```

Returns

entity, attribute, kwargs - the values supplied when the callback was initially created.

Parameters

handle

The handle returned when the `listen_state()` call was made.

Examples

```
entity, attribute, kwargs = self.info_listen_state(self.handle)
```

5.2 Scheduler Calls

Run the callback in a defined number of seconds. This is used to add a delay, for instance a 60 second delay before a light is turned off after it has been triggered by a motion detector. This callback should always be used instead of `time.sleep()` as discussed previously.

```
self.handle = self.run_in(callback, delay, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

Delay, in seconds before the callback is invoked.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
self.handle = self.run_in(self.run_in_c)
self.handle = self.run_in(self.run_in_c, title = "run_in5")
```

Run the callback once, at the specified time of day. If the time of day is in the past, the callback will occur on the next day.

```
self.handle = self.run_once(callback, time, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `time` object that specifies when the callback will occur. If the time specified is in the past, the callback will occur the next day at the specified time.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run at 4pm today, or 4pm tomorrow if it is already after 4pm
import datetime
...
runtime = datetime.time(16, 0, 0)
handle = self.run_once(self.run_once_c, runtime)
```

Run the callback once, at the specified date and time.

```
self.handle = self.run_at(callback, datetime, **kwargs)
```

A handle that can be used to cancel the timer. `run_at()` will raise an exception if the specified time is in the past.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `datetime` object that specifies when the callback will occur.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run at 4pm today
import datetime
...
runtime = datetime.time(16, 0, 0)
today = datetime.date.today()
event = datetime.datetime.combine(today, runtime)
handle = self.run_once(self.run_once_c, event)
```

Execute a callback at the same time every day. If the time has already passed, the function will not be invoked until the following day at the specified time.

```
self.handle = self.run_daily(callback, start, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `time` object that specifies when the callback will occur. If the time specified is in the past, the callback will occur the next day at the specified time.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run daily at 7pm
import datetime
...
time = datetime.time(19, 0, 0)
self.run_daily(self.run_daily_c, runtime)
```

Execute a callback at the same time every hour. If the time has already passed, the function will not be invoked until the following hour at the specified time.

```
self.handle = self.run_hourly(callback, start, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `time` object that specifies when the callback will occur, the hour component of the time object is ignored. If the time specified is in the past, the callback will occur the next hour at the specified time. If time is not supplied, the callback will start an hour from the time that `run_hourly()` was executed.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run every hour, on the hour
import datetime
...
time = datetime.time(0, 0, 0)
self.run_hourly(self.run_hourly_c, runtime)
```

Execute a callback at the same time every minute. If the time has already passed, the function will not be invoked until the following minute at the specified time.

```
self.handle = self.run_minutely(callback, start, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `time` object that specifies when the callback will occur, the hour and minute components of the time object are ignored. If the time specified is in the past, the callback will occur the next hour at the specified time. If time is not supplied, the callback will start a minute from the time that `run_minutely()` was executed.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run Every Minute on the minute
import datetime
...
time = datetime.time(0, 0, 0)
self.run_minutely(self.run_minutely_c, time)
```

Execute a repeating callback with a configurable delay starting at a specific time.

```
self.handle = self.run_every(callback, time, repeat, **kwargs)
```

A handle that can be used to cancel the timer.

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

A Python `datetime` object that specifies when the initial callback will occur.

After the initial callback has occurred, another will occur every `repeat` seconds.

Arbitary keyword parameters to be provided to the callback function when it is invoked.

```
 Run every 17 minutes starting in 2 hours time
import datetime
...
self.run_every(self.run_every_c, time, 17 * 60)
```

Cancel a previously created timer

```
self.cancel_timer(handle)
```

None

A handle value returned from the original call to create the timer.

```
self.cancel_timer(handle)
```

5.2.1 info_timer()

Get information on a scheduler event from it's handle.

Synopsis

```
time, interval, kwargs = self.info_timer(handle)
```

Returns

time - datetime object representing the next time the callback will be fired

interval - repeat interval if applicable, `0` otherwise.

kwargs - the values supplied when the callback was initially created.

Parameters

handle

The handle returned when the scheduler call was made.

Examples

```
time, interval, kwargs = self.info_timer(handle)
```

5.3 Sunrise and Sunset

5.3.1 run_at_sunrise()

Run a callback every day at or around sunrise.

Synopsis

```
self.handle = self.run_at_sunrise(callback, **kwargs)
```

Returns

A handle that can be used to cancel the timer.

Parameters

callback

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

offset =

The time in seconds that the callback should be delayed after sunrise. A negative value will result in the callback occurring before sunrise. This parameter cannot be combined with `random_start` or `random_end`

**kwargs

Arbitary keyword parameters to be provided to the callback function when it is invoked.

Examples

```
import datetime
...
 Run 45 minutes before sunset
self.run_at_sunrise(self.sun, offset = datetime.timedelta(minutes = -45).total_
↪seconds(), "Sunrise -45 mins")
 or you can just do the math yourself
self.run_at_sunrise(self.sun, offset = 30 * 60, "Sunrise +30 mins")
 Run at a random time +/- 60 minutes from sunrise
self.run_at_sunrise(self.sun, random_start = -60*60, random_end = 60*60, "Sunrise,␣
↪random +/- 60 mins")
 Run at a random time between 30 and 60 minutes before sunrise
self.run_at_sunrise(self.sun, random_start = -60*60, random_end = 30*60, "Sunrise,␣
↪random - 30 - 60 mins")
```

5.3.2 run_at_sunset()

Run a callback every day at or around sunset.

Synopsis

```
self.handle = self.run_at_sunset(callback, **kwargs)
```

Returns

A handle that can be used to cancel the timer.

Parameters

callback

Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented above.

offset =

The time in seconds that the callback should be delayed after sunrise. A negative value will result in the callback occurring before sunrise. This parameter cannot be combined with `random_start` or `random_end`

**kwargs

Arbitary keyword parameters to be provided to the callback function when it is invoked.

Examples

```
 Example using timedelta
import datetime
...
self.run_at_sunset(self.sun, offset = datetime.timedelta(minutes = -45).total_
↪seconds(), "Sunset -45 mins")
 or you can just do the math yourself
self.run_at_sunset(self.sun, offset = 30 * 60, "Sunset +30 mins")
 Run at a random time +/- 60 minutes from sunset
self.run_at_sunset(self.sun, random_start = -60*60, random_end = 60*60, "Sunset,␣
↪random +/- 60 mins")
 Run at a random time between 30 and 60 minutes before sunset
self.run_at_sunset(self.sun, random_start = -60*60, random_end = 30*60, "Sunset,␣
↪random - 30 - 60 mins")
```

5.3.3 sunrise()

Return the time that the next Sunrise will occur.

Synopsis

```
self.sunrise()
```

Returns

A Python datetime that represents the next time Sunrise will occur.

Examples

```
rise_time = self.sunrise()
```

5.3.4 sunset()

Return the time that the next Sunset will occur.

Synopsis

```
self.sunset()
```

Returns

A Python datetime that represents the next time Sunset will occur.

Examples

```
set_time = self.sunset()
```

5.3.5 sun_up()

A function that allows you to determine if the sun is currently up.

Synopsis

```
result = self.sun_up()
```

Returns

`True` if the sun is up, False otherwise.

Examples

```
if self.sun_up():
    do something
```

5.3.6 sun_down()

A function that allows you to determine if the sun is currently down.

Synopsis

```
result = self.sun_down()
```

Returns

`True` if the sun is down, False otherwise.

Examples

```
if self.sun_down():
    do something
```

5.4 Services

5.4.1 call_service()

Call service is the basic way of calling a service within AppDaemon. It can call any service and provide any required parameters. Available services can be found using the developer tools in the UI. For listed services, the part before the first period is the domain, and the part after is the service name. For instance, `light.turn_on` has a domain of `light` and a service name of `turn_on`.

Synopsis

```
self.call_service(self, service, **kwargs)
```

Returns

None

Parameters

service

The service name, e.g. `light.turn_on`.

**kwargs

Each service has different parameter requirements. This argument allows you to specify a comma separated list of keyword value pairs, e.g. `entity_id = light.office_1`. These parameters will be different for every service and can be discovered using the developer tools. Most if not all service calls require an `entity_id` however, so use of the above example is very common with this call.

Examples

```
self.call_service("light/turn_on", entity_id = "light/office_lamp", color_name = "red
↪")
self.call_service("notify/notify", title = "Hello", message = "Hello World")
```

5.4.2 turn_on()

This is a convenience function for the `homassistant.turn_on` function. It is able to turn on pretty much anything in Home Assistant that can be turned on or run:

- Lights
- Switches
- Scenes
- Scripts

And many more.

Synopsis

```
self.turn_on(entity_id, **kwargs)
```

Returns

None

Parameters

entity_id

Fully qualified entity_id of the thing to be turned on, e.g. `light.office_lamp` or `scene.downstairs_on`

**kwargs

A comma separated list of key value pairs to allow specification of parameters over and above `entity_id`.

Examples

```
self.turn_on("switch.patio_lights")
self.turn_on("scene.bedrrom_on")
self.turn_on("light.office_1", color_name = "green")
```

5.4.3 turn_off()

This is a convenience function for the `homassistant.turn_off` function. Like `homeassistant.turn_on`, it is able to turn off pretty much anything in Home Assistant that can be turned off.

Synopsis

```
self.turn_off(entity_id)
```

Returns

None

Parameters

entity_id

Fully qualified entity_id of the thing to be turned off, e.g. `light.office_lamp` or `scene.downstairs_on`.

Examples

```
self.turn_off("switch.patio_lights")
self.turn_off("light.office_1")
```

5.4.4 toggle()

This is a convenience function for the `homassistant.toggle` function. It is able to flip the state of pretty much anything in Home Assistant that can be turned on or off.

Synopsis

```
self.toggle(entity_id)
```

Returns

None

Parameters

entity_id

Fully qualified entity_id of the thing to be toggled, e.g. `light.office_lamp` or `scene.downstairs_on`.

Examples

```
self.toggle("switch.patio_lights")
self.toggle("light.office_1", color_name = "green")
```

5.4.5 select_value()

This is a convenience function for the `input_number.set_value` function. It is able to set the value of an input_number in Home Assistant.

Synopsis

```
self.select_value(entity_id, value)
```

Returns

None

Parameters

entity_id

Fully qualified entity_id of the input_slider to be changed, e.g. `input_slider.alarm_hour`.

value

The new value to set the input slider to.

Examples

```
self.select_value("input_slider.alarm_hour", 6)
```

5.4.6 select_option()

This is a convenience function for the `input_select.select_option` function. It is able to set the value of an input_select in Home Assistant.

Synopsis

```
self.select_option(entity_id, option)
```

Returns

None

Parameters

entity_id

Fully qualified entity_id of the input_select to be changed, e.g. `input_select.mode`.

value

The new value to set the input slider to.

Examples

```
self.select_option("input_select.mode", "Day")
```

5.4.7 notify()

This is a convenience function for the `notify.notify` service. It will send a notification to a named notification service. If the name is not specified it will default to `notify/notify`.

Synopsis

```
notify(message, **kwargs)
```

Returns

None

Parameters

message

Message to be sent to the notification service.

title =

Title of the notification - optional.

name =

Name of the notification service - optional.

Examples

```
self.notify("", "Switching mode to Evening")
self.notify("Switching mode to Evening", title = "Some Subject", name = "smtp")
```

5.5 Events

5.5.1 listen_event()

Listen event sets up a callback for a specific event, or any event.

Synopsis

```
handle = listen_event(function, event = None, **kwargs):
```

Returns

A handle that can be used to cancel the callback.

Parameters

function

The function to be called when the event is fired.

event

Name of the event to subscribe to. Can be a standard Home Assistant event such as `service_registered` or an arbitrary custom event such as `"MODE_CHANGE"`. If no event is specified, `listen_event()` will subscribe to all events.

**kwargs (optional)

One or more keyword value pairs representing App specific parameters to supply to the callback. If the keywords match values within the event data, they will act as filters, meaning that if they don't match the values, the callback will not fire.

As an example of this, a Minimote controller when activated will generate an event called `zwave.scene_activated`, along with 2 pieces of data that are specific to the event - `entity_id` and `scene`. If you include keyword values for either of those, the values supplied to the 'listen_event()1 call must match the values in the event or it will not fire. If the keywords do not match any of the data in the event they are simply ignored.

Filtering will work with any event type, but it will be necessary to figure out the data associated with the event to understand what values can be filtered on. This can be achieved by examining Home Assistant's logfiles when the event fires.

Examples

```
self.listen_event(self.mode_event, "MODE_CHANGE")
 Listen for a minimote event activating scene 3:
self.listen_event(self.generic_event, "zwave.scene_activated", scene_id = 3)
 Listen for a minimote event activating scene 3 from a specific minimote:
self.listen_event(self.generic_event, "zwave.scene_activated", entity_id = "minimote_
↪31", scene_id = 3)
```

5.5.2 cancel_listen_event()

Cancels callbacks for a specific event.

Synopsis

```
cancel_listen_event(handle)
```

Returns

None.

Parameters

handle

A handle returned from a previous call to `listen_event()`.

Examples

```
self.cancel_listen_event(handle)
```

5.5.3 info_listen_event()

Get information on an event callback from it's handle.

Synopsis

```
service, kwargs = self.info_listen_event(handle)
```

Returns

service, kwargs - the values supplied when the callback was initially created.

Parameters

handle

The handle returned when the `listen_event()` call was made.

Examples

```
service, kwargs = self.info_listen_event(handle)
```

5.5.4 fire_event()

Fire an event on the HomeAssistant bus, for other components to hear.

Synopsis

```
fire_event(event, **kwargs)
```

Returns

None.

Parameters

event

Name of the event. Can be a standard Home Assistant event such as `service_registered` or an arbitrary custom event such as `"MODE_CHANGE"`.

**kwargs

Zero or more keyword arguments that will be supplied as part of the event.

Examples

```
self.fire_event("MY_CUSTOM_EVENT", jam="true")
```

5.6 Presence

5.6.1 get_trackers()

Return a list of all device tracker names. This is designed to be iterated over.

Synopsis

```
tracker_list = get_trackers()
```

Returns

An iterable list of all device trackers.

Examples

```
trackers = self.get_trackers()
for tracker in trackers:
    do something
```

5.6.2 get_tracker_details()

Return a list of all device trackers and their associated state.

Synopsis

```
tracker_list = get_tracker_details()
```

Returns

A list of all device trackers with their associated state.

Examples

```
trackers = self.get_tracker_details()
for tracker in trackers:
    do something
```

5.6.3 get_tracker_state()

Get the state of a tracker. The values returned depend in part on the configuration and type of device trackers in the system. Simpler tracker types like `Locative` or `NMAP` will return one of 2 states:

- `home`
- `not_home`

Some types of device tracker are in addition able to supply locations that have been configured as Geofences, in which case the name of that location can be returned.

Synopsis

```
location = self.get_tracker_state(tracker_id)
```

Returns

A string representing the location of the tracker.

Parameters

tracker_id

Fully qualified entity_id of the device tracker to query, e.g. `device_tracker.andrew`.

Examples

```
trackers = self.get_trackers()
for tracker in trackers:
  self.log("{} is {}".format(tracker, self.get_tracker_state(tracker)))
```

5.6.4 everyone_home()

A convenience function to determine if everyone is home. Use this in preference to getting the state of `group.all_devices()` as it avoids a race condition when using state change callbacks for device trackers.

Synopsis

```
result = self.everyone_home()
```

Returns

Returns `True` if everyone is at home, `False` otherwise.

Examples

```
if self.everyone_home():
    do something
```

5.6.5 anyone_home()

A convenience function to determine if one or more person is home. Use this in preference to getting the state of `group.all_devices()` as it avoids a race condition when using state change callbacks for device trackers.

Synopsis

```
result = self.anyone_home()
```

Returns

Returns `True` if anyone is at home, `False` otherwise.

Examples

```
if self.anyone_home():
    do something
```

5.6.6 noone_home()

A convenience function to determine if no people are at home. Use this in preference to getting the state of group.all_devices() as it avoids a race condition when using state change callbacks for device trackers.

Synopsis

```
result = self.noone_home()
```

Returns

Returns `True` if no one is home, `False` otherwise.

Examples

```
if self.noone_home():
    do something
```

5.7 Miscellaneous Helper Functions

5.7.1 time()

Returns a python `time` object representing the current time. Use this in preference to the standard Python ways to discover the current time, especially when using the "Time Travel" feature for testing.

Synopsis

```
time()
```

Returns

A localised Python time object representing the current AppDaemon time.

Parameters

None

Example

```
now = self.time()
```

5.7.2 date()

Returns a python `date` object representing the current date. Use this in preference to the standard Python ways to discover the current date, especially when using the "Time Travel" feature for testing.

Synopsis

```
date()
```

Returns

A localised Python time object representing the current AppDaemon date.

Parameters

None

Example

```
today = self.date()
```

5.7.3 datetime()

Returns a python `datetime` object representing the current date and time. Use this in preference to the standard Python ways to discover the current time, especially when using the "Time Travel" feature for testing.

Synopsis

```
datetime()
```

Returns

A localised Python datetime object representing the current AppDaemon date and time.

Parameters

None

Example

```
now = self.datetime()
```

5.7.4 convert_utc()

Home Assistant provides timestamps of several different sorts that may be used to gain additional insight into state changes. These timestamps are in UTC and are coded as ISO 8601 Combined date and time strings. `convert_utc()` will accept one of these strings and convert it to a localised Python datetime object representing the timestamp

Synopsis

```
convert_utc(utc_string)
```

Returns

`convert_utc(utc_string)` returns a localised Python datetime object representing the timestamp.

Parameters

utc_string

An ISO 8601 encoded date and time string in the following format: `2016-07-13T14:24:02.040658-04:00`

Example

5.7.5 parse_time()

Takes a string representation of a time, or sunrise or sunset offset and converts it to a `datetime.time` object.

Synopsis

```
parse_time(time_string)
```

Returns

A `datetime.time` object, representing the time given in the `time_string` argument.

Parameters

time_string

A representation of the time in a string format with one of the following formats:

- HH:MM:SS - the time in Hours Minutes and Seconds, 24 hour format.
- sunrise|sunset [+|- HH:MM:SS]- time of the next sunrise or sunset with an optional positive or negative offset in Hours Minutes and seconds

Example

```
time = self.parse_time("17:30:00")
time = self.parse_time("sunrise")
time = self.parse_time("sunset + 00:30:00")
time = self.parse_time("sunrise + 01:00:00")
```

5.7.6 now_is_between()

Takes two string representations of a time, or sunrise or sunset offset and returns true if the current time is between those 2 times. `now_is_between()` can correctly handle transitions across midnight.

Synopsis

```
now_is_between(start_time_string, end_time_string)
```

Returns

`True` if the current time is within the specified start and end times, `False` otherwise.

Parameters

start_time_string, end_time_string

A representation of the start and end time respectively in a string format with one of the following formats:

- HH:MM:SS - the time in Hours Minutes and Seconds, 24 hour format.
- `sunrise|sunset` [+|- HH:MM:SS]- time of the next sunrise or sunset with an optional positive or negative offset in Hours Minutes and seconds

Example

```
if self.now_is_between("17:30:00", "08:00:00"):
    do something
if self.now_is_between("sunset - 00:45:00", "sunrise + 00:45:00"):
    do something
```

5.7.7 friendly_name()

`frindly_name()` will return the Friendly Name of an entity if it has one.

Synopsis

```
Name = self.friendly_name(entity_id)
```

Returns

The friendly name of the entity if it exists or the entity id if not.

Example

```
tracker = "device_tracker.andrew"
self.log("{}  ({}) is {}".format(tracker, self.friendly_name(tracker), self.get_
↪tracker_state(tracker)))
```

5.7.8 split_entity()

`split_entity()` will take a fully qualified entity id of the form `light.hall_light` and split it into 2 values, the device and the entity, e.g. `light` and `hall_light`.

Synopsis

```
device, entity = self.split_entity(entity_id)
```

Parameters

entity_id

Fully qualified entity id to be split.

Returns

A list with 2 entries, the device and entity respectively.

Example

```
device, entity = self.split_entity(entity_id)
if device == "scene":
    do something specific to scenes
```

5.7.9 entity_exists()

Synopsis

```
entity_exists(entity)
```

`entity_exists()` is used to verify if a given entity exists in Home Assistant or not.

Returns

`entity_exists()` returns `True` if the entity exists, `False` otherwise.

Parameters

entity

The fully qualified name of the entity to check for (including the device type)

Examples

```
 Return state for the entire system
if self.entity_exists("light.living_room"):
  do something
  ...
```

5.7.10 get_app()

`get_app()` will return the instantiated object of another app running within the system. This is useful for calling functions or accessing variables that reside in different apps without requiring duplication of code.

Synopsis

```
get_app(self, name)
```

Parameters

name

Name of the app required. This is the name specified in header section of the config file, not the module or class.

Returns

An object reference to the class.

Example

```
MyApp = self.get_app("MotionLights")
MyApp.turn_light_on()
```

5.7.11 split_device_list()

`split_device_list()` will take a comma separated list of device types (or anything else for that matter) and return them as an iterable list. This is intended to assist in use cases where the App takes a list of entities from an argument, e.g. a list of sensors to monitor. If only one entry is provided, an iterable list will still be returned to avoid the need for special processing.

Synopsis

```
devices = split_device_list(list)
```

Returns

A list of split devices with 1 or more entries.

Example

```
for sensor in self.split_device_list(self.args["sensors"]):
    do something for each sensor, e.g. make a state subscription
```

5.8 Logfiles

5.8.1 log()

Synopsis

```
log(message, level = "INFO")
```

Returns

Nothing

Parameters

Message

The message to log.

level

The log level of the message - takes a string representing the standard logger levels.

Examples

```
self.log("Log Test: Parameter is {}".format(some_variable))
self.log("Log Test: Parameter is {}".format(some_variable), level = "ERROR")
self.log("Line: __line__, module: __module__, function: __function__, Message:␣
↪Something bad happened")
```

5.8.2 error()

Synopsis

```
error(message, level = "WARNING")
```

Returns

Nothing

Parameters

Message

The message to log.

level

The log level of the message - takes a string representing the standard logger levels.

Examples

```
self.error("Some Warning string")
self.error("Some Critical string", level = "CRITICAL")
```

5.9 API

5.9.1 register_endpoint()

Register an endpoint for API calls into an App.

Synopsis

```
register_endpoint(callback, name = None)
```

Returns

handle - a handle that can be used to remove the registration

Parameters

callback

The function to be called when a request is made to the named endpoint

name

The name of the endpoint to be used for the call. If `None` the name of the App will be used.

Examples

```
self.register_endpoint(my_callback)
self.register_callback(alexa_cb, "alexa")
```

5.9.2 unregister_endpoint()

Remove a previously registered endpoint.

Synopsis

```
unregister_endpoint(handle)
```

Returns

None

Parameters

handle

A handle returned by a previous call to `register_endpoint`

Examples

```
self.unregister_endpoint(handle)
```

5.10 Alexa Helper Functions

5.10.1 get_alexa_intent()

Register an endpoint for API calls into an App.

Synopsis

```
self.get_alexa_intent(data)
```

Returns

A string representing the Intent from the interaction model that was requested

Parameters

data

The request data received from Alexa.

Examples

```
intent = self.get_alexa_intent(data)
```

5.10.2 get_alexa_slot_value()

Return values for slots form the interaction model.

Synopsis

```
self.get_alexa_slot_value(data, name = None)
```

Returns

A string representing the value of the slot from the interaction model, or a hash of slots.

Parameters

data

The request data received from Alexa.

name

Name of the slot. If a name is not specified, all slots will be returned as a dictionary. If a name is spedicied but is not found, `None` will be returned.

Examples

```
beer_type = self.get_alexa_intent(data, "beer_type")
all_slots = self.get_alexa_intent(data)
```

self.format_alexa_response(speech = speech, card = card, title = title)

5.10.3 format_alexa_response()

Format a response to be returned to Alex including speech and a card.

Synopsis

```
self.format_alexa_response(speech = speech, card = card, title = title)
```

Returns

None

Parameters

speech =

The text for Alexa to say

card =

Text for the card

title =

Title for the card

Examples

```
format_alexa_response(speech = "Hello World", card = "Greetings to the world", title
↪= "Hello")
```

5.11 Google Home Helper Functions

5.11.1 get_apiai_intent()

Register an endpoint for API calls into an App.

Synopsis

```
self.get_apiai_intent(data)
```

Returns

A string representing the Intent from the interaction model that was requested

Parameters

data

The request data received from Google Home.

Examples

```
intent = self.get_apiai_intent(data)
```

5.11.2 get_apiai_slot_value()

Return values for slots form the interaction model.

Synopsis

```
self.get_apiai_slot_value(data, name = None)
```

Returns

A string representing the value of the slot from the interaction model, or a hash of slots.

Parameters

data

The request data received from Google Home.

name

Name of the slot. If a name is not specified, all slots will be returned as a dictionary. If a name is spedicied but is not found, `None` will be returned.

Examples

```
beer_type = self.get_apiai_intent(data, "beer_type")
all_slots = self.get_apiai_intent(data)
```

self.format_apiai_response(speech = speech)

5.11.3 format_appapi_response()

Format a response to be returned to Google Home including speech.

Synopsis

```
self.format_apiai_response(speech = speech)
```

Returns

None

Parameters

speech =

The text for Google Home to say

Examples

```
format_apiai_response(speech = "Hello World")
```

5.12 Dashboard Functions

5.12.1 set_app_state()

Publish state information to AppDaemon's internal state and push the state changes out to listening Apps and Dashboards.

Synopsis

```
self.set_app_state(entity_id, state)
```

Returns

None.

Parameters

entity_id

A name for the new state. It must conform to the standard entity_id format, e.g. `<device_type>.<name>`. however device type and name can be whatever you like as long as you ensure it doesn't conflict with any real devices. For clarity, I suggest the convention of using `appdaemon` as the device type. A single App can publish to as many entity ids as desired.

state

The state to be associated with the entity id. This is a dictionary and must contain the enirety of the state information, It will replace the old state information, and calls like `listen_state()` should work correctly reporting the old and the new state information as long as you keep the dictionary looking similar to HA status updates, e.g. the main state in a state field, and any attibutes in an attributes sub-dictionary.

Examples

```
self.set_app_state("appdaemon.alerts", {"state": number, "attributes": {"unit_of_
↪measurement": ""}})
```

This is an example of a state update that can be used with a sensor widget in HADashboard. "state" is the actual value, and the widget also expects an attribute called "unit_of_measurement" to work correctly.

5.12.2 dash_navigate()

Force all connected Dashboards to navigate to a new URL

Synopsis

```
dash_navigate(self, target, timeout = -1, ret = None)
```

Returns

None.

Parameters

target

A URL for the dashboard to navigate to e.g. `/MainDash`

ret

Time to wait before the optional second change. If not specified the first change will be permanent.

timeout

URL to navigate back to after `timeout`. If not specified, the dashboard will navigate back to the original panel.

Examples

```
self.dash_navigate("/AlarmStatus", timeout=10)        # Switch to AlarmStatus Panel␣
↪then return to current panel after 10 seconds
self.dash_navigate("/Locks", timeout=10, ret="/Main") # Switch to Locks Panel then␣
↪return to Main panel after 10 seconds
```

CHAPTER 6

Dashboard Install and Configuration

HADashboard is a dashboard for Home Assistant that is intended to be wall mounted, and is optimized for distance viewing.

6.1 Installation and Configuration

HADashboard is dependent upon AppDaemon. As a first step please refer to the AppDaemon Installation Documentation.

When you have AppDaemon installed and running, configuration of the Dashboard is pretty simple. You just need to add a directive to the config file - `dash_url`.

This and the optional `dash_dir` directive should be in the top of the file under a new `HADashboard:` section.

- `dash_url` - the url you want the dashboard service to listen on

For instance:

```
HASS:
  ha_url: <some_url>
  ...
HADashboard:
  dash_url: http://192.168.1.20:5050
```

To enable https support for HADashboard, add the following directives pointing to your certificate and keyfile:

```
dash_ssl_certificate: /etc/letsencrypt/live/somehost/fullchain.pem
dash_ssl_key: /etc/letsencrypt/live/somehost/privkey.pem
```

To password protect HADashboard use the `dash_password` directive:

```
dash_password: some_password
```

Or you can use the secret function and place the actual password in your secrets.yaml file:

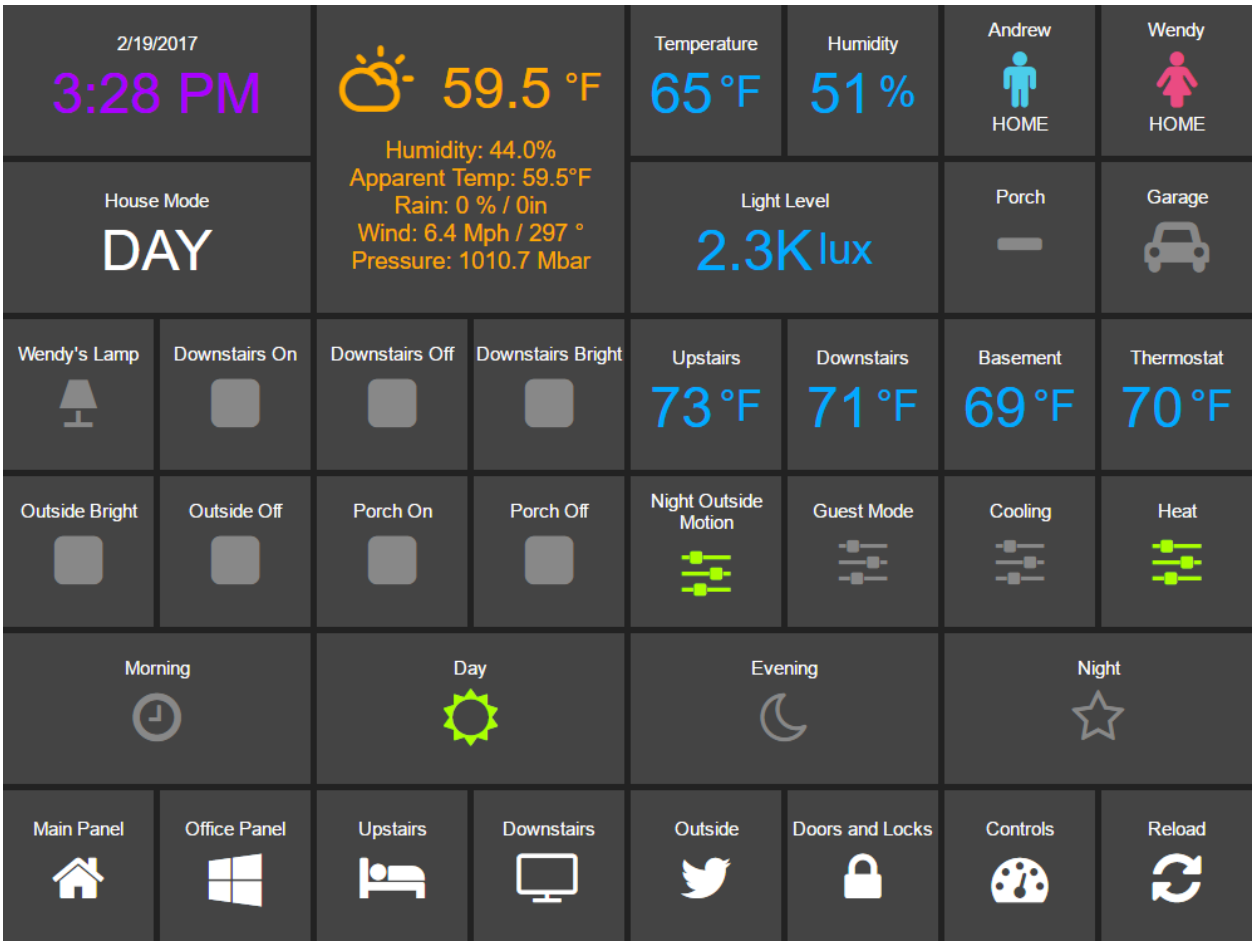
2/19/2017
3:28 PM
59.5 °F
Humidity: 44.0%
Apparent Temp: 59.5°F
Rain: 0 % / 0in
Wind: 6.4 Mph / 297 °
Pressure: 1010.7 Mbar
Temperature
65 °F
Humidity
51 %
Andrew
HOME
Wendy
HOME
House Mode
DAY
Light Level
2.3K lux
Porch
Garage
Wendy's Lamp
Downstairs On
Downstairs Off
Downstairs Bright
Upstairs
73 °F
Downstairs
71 °F
Basement
69 °F
Thermostat
70 °F
Outside Bright
Outside Off
Porch On
Porch Off
Night Outside Motion
Guest Mode
Cooling
Heat
Morning
Day
Evening
Night
Main Panel
Office Panel
Upstairs
Downstairs
Outside
Doors and Locks
Controls
Reload

Fig. 6.1: UI

```
dash_password: !secret dash_password
```

By default, dashboards are searched for under the config directory in a sub directory called `dashboards`. Optionally, you can place your dashboards in a directory other than under the config directory using the `dash_dir` directive.

e.g.:

```
HADashboard:
dash_dir: /etc/appdaemon/dashboards
```

Next, you will need to create the `dashboards` directive either under the conf directory, or wherever you specify with `dash_dir`. Once that is done, for testing purposes, create a file in the dashboards directory called `Hello.dash` and paste in the following:

```
##
## Main arguments, all optional
##
title: Hello Panel
widget_dimensions: [120, 120]
widget_margins: [5, 5]
columns: 8

label:
    widget_type: label
    text: Hello World

layout:
    - label(2x2)
```

When you have added the lines to the config and created the dashboards directory and test dashbboard, restart AppDaemon and you will be ready to go. If you navigate to the top level, e.g. `http://192.168.1.20:5050` in the case above, you will see a welcome page with a list of configured dashboards. If you haven't yet configured any the list will be empty.

When you have created a dashboard you can navigate to it by going to `http://192.168.1.20:5050/<Dashboard Name>`

If you are using AppDaemon just for the dasboard and not the Apps, you can disable the app engine with the following directive:

```
AppDaemon:
  disable_apps: 1
```

This will free up some CPU and memory.

HADashboard pre-compiles all of the user created Dashboard for efficiency. It will detect when changes have been made to widgets, styles or dahsboards and automatically recompile. This is usually desirable as compilation can take several seconds on slower hardware for a fully loaded dashboard, however to force a recompilation every time, use the following directive:

```
HADashboard:
dash_force_compile: 1
```

This will force dashboard recompilation whenever the dashboard is loaded. You can also force a recompilation by adding the parameter `recompile=1` to the dashboard URL.

By default, information and errors around access to the Dashboard will go to the same place as AppDaemon's log. To split the page access out to a different file, use the `accessfile` directive, e.g.:

```
HADashboard:
accessfile: /var/log/dash_access
```

To force dashboard recompilation of all dashboards after a restart, use:

```
HADashboard:
dash_compile_on_start: 1
```

This should not be necessary but may on occasion be required after an upgrade to pickup changes.

6.2 Dashboard URL Parameters

The dashboard URL supports a couple of extra parameters:

- `skin` - name of the skin you want to use, default is `default`
- `recompile` - set to anything to force a recompilation of the dashboard

For example, the following url will load a dasboard called main with the obsidian skin:

```
http://<ip address>:<port>/Main?skin=obsidian
```

CHAPTER 7

Dashboard Creation

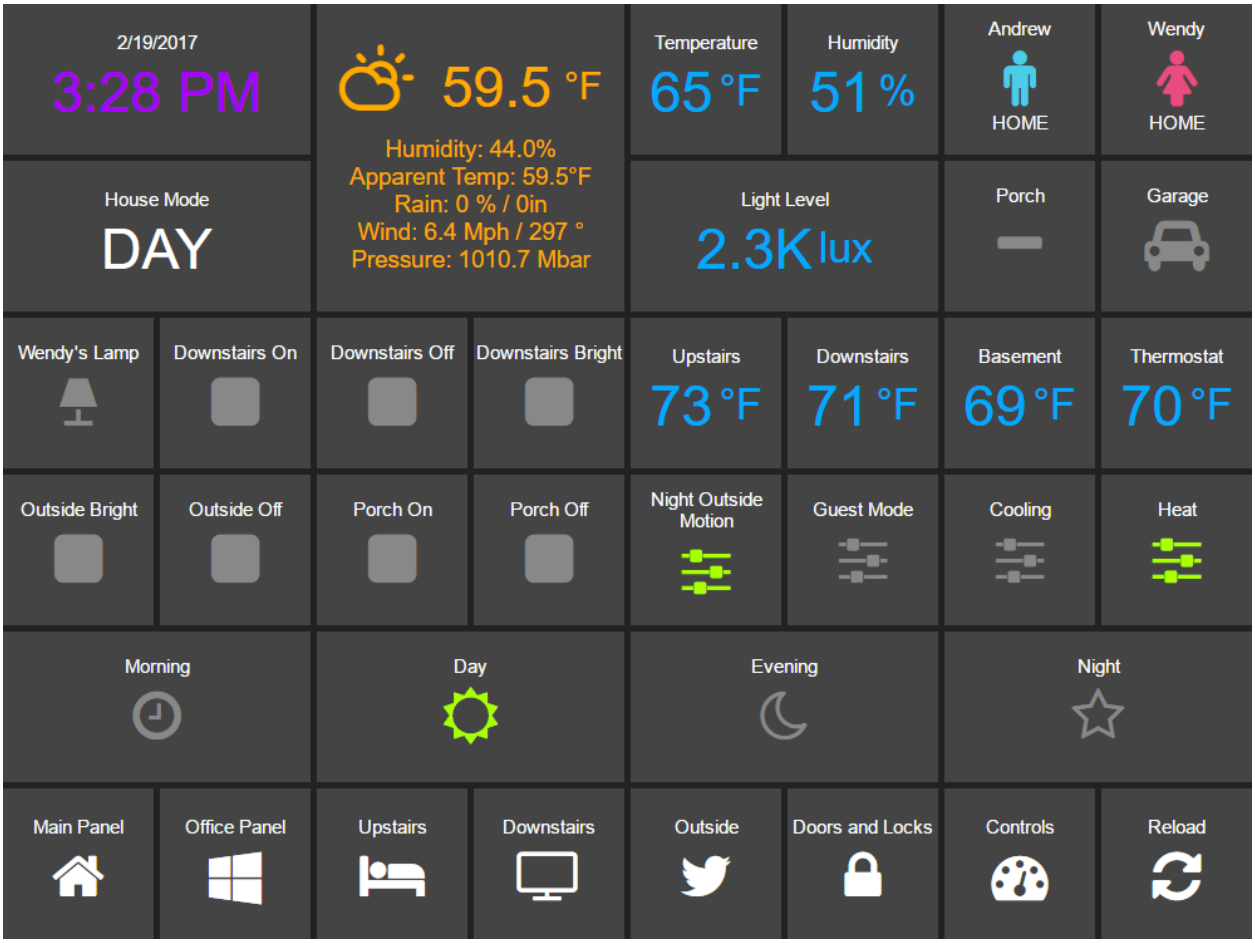

Fig. 7.1: UI

7.1 Dashboard Creation

Dashboard configuration is simple yet very powerful. Dashboards can be created in single files or made modular for reuse of blocks of widgets. Dashboards are configured using YAML.

We will start with a simple single file configuration. Create a file with a `.dash` extension in the `dashboards` directory, and pull it up in your favorite editor.

7.1.1 Main Settings

A top level dashboard will usually have one of a number of initial directives to configure aspects of the dashboard, although they are all optional. An example is as follows:

```
##
## Main arguments, all optional
##
title: Main Panel
widget_dimensions: [120, 120]
widget_size: [1, 1]
widget_margins: [5, 5]
columns: 8
global_parameters:
    use_comma: 0
    precision: 1
    use_hass_icon: 1
```

These are all fairly self explanatory:

- `title` - the name that will end up in the title of the web page, defaults to "HADashboard".
- `widget_dimensions` - the unit height and width of the individual widgets in pixels. Note that the absolute size is not too important as on tablets at least the browser will scale the page to fit. What is more important is the aspect ratio of the widgets as this will affect whether or not the dashboard completely fills the tablets screen. The default is [120, 120] (width, height). This works well for a regular iPad.
- `widget_size` - the number of grid blocks each widget will be by default if not specified
- `widget_margins` - the size of blank space between widgets.
- `rows` - the total number of rows in the dashboard. This will help with spacing, but is optional for dashboards with fewer than 15 rows
- `columns` - the number of columns the dasboard will have.
- `global_parameters` - a list of parameters that will be applied to every widget. If the widget does not accept that parameter it will be ignored. Global parameters can be overriden at the widget definition if desired. This is useful for instance if you want to use commas as decimals for all of your widgets. This will also apply to widgets defined with just their entity ids so they will not require a formal widget definition just to change the decimal separator.

The very simplest dashboard needs a layout so it can understand where to place the widgets. We use a `layout` directive to tell HADasboard how to place them. Here is an example:

```
layout:
    - light.hall, light.living_room, input_boolean.heating
    - media_player(2x1), sensor.temperature
```

As you can see, here we are refering directly to native Home Assistant entities. From this, HADashboard is able to figure out the right widget type and grab it's friendly name and add it to the dasboard. For the `clock` and `weather` widgets there is no associated entity id so just your `clock.clock` or `weather.weather`.

The layout command is intended to be visual in how you lay out the widgets. Each layout entry represents a row on the dashboard, each comma separated widget represents a cell on that row.

Widgets can also have a size associated with them - that is the `(2x1)` directive appended to the name. This is simply the width of the widget in columns and the height of the widget in rows. For instance, `(2x1)` would refer to a widget 2 cells wide and 1 cell high. If you leave of the sizing information, the widget will use the `widget_size` dashboard parameter if specified, or default to `(1x1)` if not. HADasboard will do it's best to calculate the right layout from what you give it but expect strange behavior if you add too many widgets on a line.

For a better visual cue you can lay the widgets out with appropriate spacing to see what the grid will look like more intuitively:

```
layout:
   - light.hall,       light.living_room, input_boolean.heating
   - media_player(2x1),                   sensor.temperature
```

... and so on.

Make sure that the number of widths specified adds up to the total number of columns, and don't forget to take into account widgets that are more than one row high (e.g. the weather widget here).

If you want a blank space you can use the special widget name `spacer`. To leave a whole row empty, just leave an entry for it with no text. For instance:

```
- light.hall, light.living_room, input_boolean.heating
-
- media_player(2x1), sensor.temperature
```

The above would leave the 2nd row empty. If you want more than one empty line use `empty` as follows":

```
- light.hall, light.living_room, input_boolean.heating
- empty: 2
- media_player(2x1), sensor.temperature
```

This would leave the 2nd and 3rd rows empty.

And that is all there to it, for a simple one file dashboard.

7.1.2 Detailed Widget Definition

The approach above is ok for simple widgets like lights, but HADashboard has a huge range of customization options. To access these, you need to formally define the widget along with its associated parameters.

To define a widget simply add lines elsewhere in the file. Give it a name , a widget type and a number of optional parameters like this:

```
weather_widget:
    widget_type: weather
    units: "&deg;F"
```

Here we have defined a widget of type "weather", and given it an optional parameter to tell it what units to use for temperature. Each widget type will have different required parameters, refer to the documentation below for a complete list for each type. All widgets support ways to customize colors and text sizes as well as attibutes they need to understand how to link the widget to Home Assistant, such as entity_ids.

Lets look at a couple more examples of widget definitions:

```
clock:
    widget_type: clock

weather:
    widget_type: weather
    units: "&deg;F"

side_temperature:
    widget_type: sensor
    title: Temperature
    units: "&deg;F"
    precision: 0
    entity: sensor.side_temp_corrected

side_humidity:
    widget_type: sensor
    title: Humidity
    units: "%"
    precision: 0
    entity: sensor.side_humidity_corrected

andrew_presence:
    widget_type: device_tracker
    title: Andrew
    device: andrews_iphone

wendy_presence:
    widget_type: device_tracker
    title: Wendy
    device: wendys_iphone

mode:
    widget_type: sensor
    title: House Mode
    entity: input_select.house_mode

light_level:
    widget_type: sensor
    title: Light Level
    units: "lux"
    precision: 0
    shorten: 1
    entity: sensor.side_multisensor_luminance_25_3

porch_motion:
    widget_type: binary_sensor
    title: Porch
    entity: binary_sensor.porch_multisensor_sensor_27_0

garage:
    widget_type: switch
    title: Garage
    entity: switch.garage_door
    icon_on: fa-car
    icon_off: fa-car
    warn: 1
```

Now, instead of an entity id we refer to the name of the widgets we just defined:

```
layout:
    - clock(2x1), weather(2x2), side_temperature(1x1), side_humidity(1x1), andrew_
↪presence(1x1), wendy_presence(1x1)
    - mode(2x1), light_level(2x1), porch_motion(1x1), garage(1x1)
```

It is also possible to add a widget from a standalone file. The file will contain a single widget definition. To create a clock widget this way we would make a file called `clock.yaml` and place it in the dashboard directory along with the dashboard. The contents would look something like this:

```
widget_type: clock
widget_style: "color: red"
```

Note that the indentation level starts at 0. To include this file, just reference a widget called `clock` in the layout, and HADashboard will automatically load the widget.

A file will override a native entity, so you can create your dashboard just using entities, but if you want to customize a specific entity, you can just create a file named `<entity_name>.yaml` and put the settings in there. You can also override entity names by specifying a widget of that name in the same or any other file, which will take priority over a standalone yaml file.

And that is all there to it, for a simple one file dashboard.

7.2 Advanced Dashboard Definition

When you get to the point where you have multiple dashboards, you may want to take a more modular approach, as you will find that in many cases you want to reuse parts of other dashboards. For instance, I have a common header for mine consisting of a row or two of widgets I want to see on every dashboard. I also have a footer of controls to switch between dashboards that I want on each dashboard as well.

To facilitate this, it is possible to include additional files, inline to build up dashboards in a more modular fashion. These additional files end in `.yaml` to distinguish them from top level dashboards. They can contain additional widget definitions and also optionally their own layouts.

The sub files are included in the layout using a variation of the layout directive:

```
layout:
    - include: top_panel
```

This will look for a file called `top_panel.yaml` in the dashboards directory, then include it. There are a couple of different ways this can be used.

- If the yaml file includes it's own layouts directive, the widgets from that file will be placed as a block, in the way described by its layout, making it reusable. You can change the order of the blocks inclusion by moving where in the original layout directive you include them.
- If the yaml file just includes widget definitions, it is possible to perform the layout in the higher level dash if you prefer so you still get an overall view of the dashboard. This approach has the benefit that you can be completely flexible in the layout wheras the first method defines fixed layouts for the included blocks.

I prefer the completely modular approach - here is an example of a full top level dashboard created in that way:

```
title: Main Panel
widget_dimensions: [120, 120]
widget_margins: [5, 5]
columns: 8
```

```
layout:
    - include: top_panel
    - include: main_middle_panel
    - include: mode_panel
    - include: bottom_panel
```

As you can see, it includes four modular sub-dashes. Since these pieces all have their own layout information there is no need for additional layout in the top level file. Here is an example of one of the self contained sub modules (mode_panel.yaml):

```
clock:
    widget_type: clock

weather:
    widget_type: weather
    units: "&deg;F"

side_temperature:
    widget_type: sensor
    title: Temperature
    units: "&deg;F"
    precision: 0
    entity: sensor.side_temp_corrected

side_humidity:
    widget_type: sensor
    title: Humidity
    units: "%"
    precision: 0
    entity: sensor.side_humidity_corrected

andrew_presence:
    widget_type: device_tracker
    title: Andrew
    device: andrews_iphone

wendy_presence:
    widget_type: device_tracker
    title: Wendy
    device: dedb5e711a24415baaae5cf8e880d852

mode:
    widget_type: sensor
    title: House Mode
    entity: input_select.house_mode

light_level:
    widget_type: sensor
    title: Light Level
    units: "lux"
    precision: 0
    shorten: 1
    entity: sensor.side_multisensor_luminance_25_3

porch_motion:
    widget_type: binary_sensor
    title: Porch
```

```
    entity: binary_sensor.porch_multisensor_sensor_27_0

garage:
    widget_type: switch
    title: Garage
    entity: switch.garage_door
    icon_on: fa-car
    icon_off: fa-car
    warn: 1

layout:
    - clock(2x1), weather(2x2), side_temperature, side_humidity, andrew_presence,␣
↪wendy_presence
    - mode(2x1), light_level(2x1), porch_motion, garage
```

Now if we take a look at that exact same layout, but assume that just the widget definitions are in the sub-blocks, we would end up with something like this - note that we must explicitly lay out each widget we have included in the other files:

```
title: Main Panel
widget_dimensions: [120, 120]
widget_margins: [5, 5]
columns: 8

layout:
    - include: top_panel
    - include: main_middle_panel
    - include: mode_panel
    - include: bottom_panel
    - clock(2x1), weather(2x2), side_temperature, side_humidity, andrew_presence,␣
↪wendy_presence
    - mode(2x1), light_level(2x1), porch_motion, garage
    - wlamp_scene, don_scene, doff_scene, dbright_scene, upstairs_thermometer,␣
↪downstairs_thermometer, basement_thermometer, thermostat_setpoint
    - obright_scene, ooff_scene, pon_scene, poff_scene, night_motion, guest_mode,␣
↪cooling, heat
    - morning(2x1), day(2x1), evening(2x1), night(2x1)
    - load_main_panel, load_upstairs_panel, load_upstairs, load_downstairs, load_
↪outside, load_doors, load_controls, reload
```

In this case, the actual layout including a widget must be after the include as you might expect.

A few caveats for loaded sub files:

- Sub files can include other subfiles to a maximum depth of 10 - please avoid circular references!
- When layout information is included in a sub file, the subfile must comprise 1 or more complete dashboard rows - partial rows or blocks are not supported.

As a final option, you can create widget definitions in the main file and use them in the layout of the header/footer/etc. For example, if you have a header that has a label in it that lists the room that the dashboard is associated with, you can put the label widget definition in the header file but all the pages get the same message. If you put the label widget definition in the main file for the room, and reference it from the layout in the header, each page has the right name displayed in the header.

For example:

```
clock:
    widget_type: clock
```

```
layout:
    - label(2x2),clock(2x2)
```

In this example of a header, we reference a clock and a label in the layout. We can re-use this header, but in order to make the label change for every page we use it on we actually define it in the dashboard file itself, and include the header in the layout:

```
title: Den Panel
widget_dimensions: [120, 120]
widget_margins: [5, 5]
columns: 8

label:
    widget_type: label
    text: Welcome to the Den

layout:
    - include: header
```

7.3 Widget Customization

Widgets allow customization using arbitary CSS styles for the individual elements that make up the widget. Every widget has a 'widget_style argument to apply styles to the whole widget, as well as one or more additional style arguments that differ for each widget. To customize a widget background for instance:

```
clock:
  widget_type: clock
  widget_style: "background: white;"
```

As is usual with CSS you can feed it multiple parameters at once, e.g.:

```
clock:
  widget_type: clock
  widget_style: "background: white; font-size: 150%;"
```

You can use any valid CSS style here although you should probably steer away from some of the formatting types as they may interact badly with HADasboards formatting. Widget level styles will correctly override just the style in the skin they are replacing.

In the case of the clock widget, it also supports date_style and time_style to modify those elements accordingly:

```
clock:
  widget_type: clock
  widget_style: "background: white"
  date_style: "color: black"
  time_style: "color: green"
```

Since date_style and time_style are applied to more specific elements, they will override widget_style. Also note that some widget styles may be specified in the widget's CSS, in which case that style will override widget_style but not the more specific styles.

7.4 State and state text

Some widgets allow you to display not only an icon showing the state but also text of the state itself. The following widgets allow this:

- scene
- binary_sensor
- switch
- device_tracker
- script
- lock
- cover
- input_boolean

In order to enable this, just add:

```
state_text: 1
```

to the widget definition. This will then make the widget show the HA state below the icon. Since native HA state is not always very pretty it is also possible to map this to better values, for instance in a different language than English.

To add a state map, just add a state_map list to the widget definition listing the HA states and what you actually want displayed. For instance:

```
state_map:
  "on": Aan
  "off": Uit
```

One wrinkle here is that YAML over enthusiastically "helps" by interpreting things like `on` and `off` as booleans so the quotes are needed to prevent this.

7.5 Icons

Widgets that allow the specification of icons have access to both Font Awesome and Material Design Icons. To specify an icon simply use the prefix `fa-` for Font Aweesome and `mdi-` for Material Design. e,g,:

```
icon_on: fa-alert
icon_off: mdi-cancel
```

In addition, the widget can be configured to use whatever icon is defined for it in Home Assistant by setting the parameter:

```
use_hass_icon: 1
```

This can also be set at the dashboard level as a global parameter.

7.6 External Commands

The dashboard can accept command from external systems to prompt actions, such as navigation to different pages. These can be achieved through a variety of means:

- AppDaemon API Calls
- HASS Automations/Scripts
- Alexa Intents

The mechanism used for this is HASS custom events. AppDaemon has it's own API calls to handle these events, for further details see the AppDaemon API Pages. The custom event name is `hadashboard` and the dashboard will respond to various commands with associated data.

To create a suitable custom event within a HASS automation, script or Alexa Intent, simply define the event and associated data as follows (this is a script example):

```
alias: Navigate
sequence:
- event: hadashboard
  event_data:
    command: navigate
    timeout: 10
    target: SensorPanel
```

The current list of commands supported and associated arguments are as follows:

7.6.1 navigate

Force any connected dashboards to navigate to a new page

Arguments

`target` - Name of the new Dashboard to navigate to, e.g. `SensorPanel` - this is not a URL. `timeout` - length of time to stay on the new dashboard before returning to the original. This argument is optional and if not specified, the navigation will be permanent.

Note that if there is a click or touch on the new panel before the timeout expires, the timeout will be cancelled.

`timeout` - length of time to stay on the new dashboard `return` - dashboard to return to after the timeout has elapsed.

7.7 Widget Reference

Here is the current list of widgets and their description and supported parameters:

7.7.1 clock

A simple 12 hour clock with the date. Not currently very customizable but it will be improved upon.

Mandatory arguments:

None

Optional Arguments:

- `time_format` - set to "24hr" if you want military time/24 hour clock
- `show_seconds` - set to 1 if you want to see seconds on the display

Style Arguments:

- `widget_style`
- `time_style`
- `date_style`

7.7.2 weather

Up to date weather reports. Requires dark sky to be configured in Home Assistant with at minimum the following sensors:

- temperature
- humidity
- precip_probability
- precip_intensity
- wind_speed
- pressure
- wind_bearing
- apparent_temperature
- icon

Mandatory arguments:

None

Optional Arguments:

None

Cosmetic Arguments:

- `widget_style`
- `main_style`
- `unit_style`
- `sub_style`

7.7.3 sensor

A widget to report on values for any sensor in Home Assistant

The widget will detect whether or not it is showing a numeric value, and if so, it will use the numeric style. If it is showing text it will use the text style, which among other things makes the text size smaller.

Mandatory Arguments:

- `entity` - the entity_id of the sensor to be monitored

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `units` - the unit symbol to be displayed, if not specified HAs unit will be used, specify "" for no units
- `precision` - the number of decimal places
- `shorten` - if set to one, the widget will abbreviate the readout for high numbers, e.g. `1.1K` instead of `1100`
- `use_comma` - if set to one', a comma will be used as the decimal separator
- `state_map`
- `sub_entity` - second entity to be displayed in the state text area
- `sub_entity_map` - state map for the sub_entity

Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `value_style`
- `text_style`
- `unit_style`

7.7.4 rss

A widget to display an RSS feed.

Note that the actual feeds are configured in appdaemon.yaml as follows:

```
AppDaemon:

  rss_feeds:
    - feed: <feed_url>
      target: <target_name>
    - feed: <feed url>
      target: <target_name>
```

```
    ...

rss_update: <feed_refresh_interval>
```

- `feed_url` - fully qualified path to rss feed, e.g. `http://rss.cnn.com/rss/cnn_topstories.rss`
- `target name` - the entity of the target RSS widget in the dashboard definition file
- `feed_refresh_interval` - how often AppDaemon will refresh the RSS feeds

There is no limit to the number of feeds you configure, and you will need to configure one RSS widget to display each feed.

The RSS news feed cannot be configured if you are still using the legacy `.cfg` file type.

Mandatory Arguments:

- `entity` - the name of the configured feed - this must match the `target_name` configured in the AppDaemon configuration
- `interval` - the period between display of different items within the feed

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `recent` - the number of most recent stories that will be shown. If not specified, all stories in the feed will be shown.

Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `text_style`

7.7.5 gauge

A widget to report on numeric values for sensors in Home Assistant in a gauge format.

Mandatory Arguments:

- `entity` - the entity_id of the sensor to be monitored
- `max` - maximum value to show
- `min` - minimum value to show

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `units` - the unit symbol to be displayed, if not specified HAs unit will be used, specify "" for no units

Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `low_color`
- `med_color`
- `high_color`
- `bgcolor`
- `color`

Note that unlike other widgets, the color settings require an actual color, rather than a CSS style.

7.7.6 device_tracker

A Widget that reports on device tracker status. It can also be optionally be used to toggle the status between "home" and "not_home".

Mandatory Arguments:

- `device` - name of the device from `known_devices.yaml`, *not* the entity_id.

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `enable` - set to 1 to enable the widget to toggle the device_tracker status
- `state_text`
- `state_map`
- `active_map`

Active map is used to specify states other than "home" that will be regarded as active, meaning the icon will light up. This can be useful if tracking a device tracker within the house using beacons for instance.

Example:

```
wendy_presence_mapped:
  widget_type: device_tracker
  title: Wendy
  title2: Mapped
  device: wendys_iphone
  active_map:
    - home
    - house
    - back_yard
    - upstairs
```

In the absence of an active map, only the state `home` will be regarded as active.

Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`
- `state_text_style`

7.7.7 label

A widget to show a simple static text string

Mandatory Arguments

None

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `text` - the text displayed on the tile

Cosmetic Arguments

- `widget_style`
- `title_style`
- `title2_style`
- `text_style`

7.7.8 scene

A widget to activate a scene

Mandatory Arguments

- `entity` - the entity_id of the scene

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.9 script

A widget to run a script

Mandatory Arguments

- `entity` - the entity_id of the script

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.10 mode

A widget to track the state of an `input_select` by showing active when it is set to a specific value. Also allows scripts to be run when activated.

Mandatory Arguments

- `entity` - the entity_id of the `input_select`
- `mode` - value of the input select to show as active
- `script` - script to run when pressed
- `state_text`
- `state_map`

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text

Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.11 switch

A widget to monitor and activate a switch

Mandatory Arguments

- `entity` - the entity_id of the switch

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Cosmetic Arguments

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.12 lock

A widget to monitor and activate a lock

Note that unlike HASS, Dashboard regards an unlocked lock as active. By contrast, the HASS UI shows a locked lock as "on". Since the purpose of the dashboard is to alert at a glance on anything that is unusual, I chose to make the unlocked state "active" which means in the default skin it is shown as red, wheras a locked icon is shown as gray. You can easily change this behavior by setting active and inactive styles if you prefer.

Mandatory Arguments

- `entity` - the entity_id of the lock

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Cosmetic Arguments

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.13 cover

A widget to monitor and activate a cover. At this time only the open and close actions are supported.

Mandatory Arguments

- `entity` - the entity_id of the cover

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Cosmetic Arguments

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.14 input_boolean

A widget to monitor and activate an input_boolean

Mandatory Arguments

- `entity` - the entity_id of the input_boolean

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Cosmetic Arguments

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.15 binary_sensor

A widget to monitor a binary_sensor

Mandatory Arguments

- `entity` - the entity_id of the binary_sensor

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

Cosmetic Arguments

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

7.7.16 light

A widget to monitor and contol a dimmable light

Mandatory Arguments

- `entity` - the entity_id of the light

Optional Arguments:

- `icon_on`
- `icon_off`
- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `on_attributes` - a list of supported HA attributes to set as initial values for the light.

Note that `rgb_color` and `xy_color` are not specified with list syntac as in Home Assistant scenes. See below for examples.

e.g.

```
testlight2:
    widget_type: light
    entity: light.office_2
    title: office_2
    on_attributes:
        brightness: 100
        color_temp: 250
```

or:

```
testlight2:
    widget_type: light
    entity: light.office_2
    title: office_2
    on_attributes:
        brightness: 100
        rgb_color: 128, 34, 56
```

or:

```
testlight2:
    widget_type: light
    entity: light.office_2
    title: office_2
    on_attributes:
        brightness: 100
        xy_color: 0.4, 0.9
```

Cosmetic Arguments

- `widget_style`

- icon_on
- icon_off
- icon_up
- icon_down
- title_style
- title2_style
- icon_style_active
- icon_style_inactive
- text_style
- level_style
- level_up_style
- level_down_style

7.7.17 input_slider

A widget to monitor and contol an input slider

Mandatory Arguments

- entity - the entity_id of the input_slider

Optional Arguments:

- title - the title displayed on the tile
- title2 - a second line of title text
- step - the size of step in brightness when fading the slider up or down
- units - the unit symbol to be displayed
- use_comma - if set to one, a comma will be used as the decimal separator

Cosmetic Arguments

- widget_style
- icon_up
- icon_down
- title_style
- title2_style
- text_style
- level_style
- level_up_style
- level_down_style

7.7.18 climate

A widget to monitor and contol a climate entity

Mandatory Arguments

- `entity` - the entity_id of the climate entity

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `step` - the size of step in temperature when fading the slider up or down
- `units` - the unit symbol to be displayed

Cosmetic Arguments

- `widget_style`
- `icon_up`
- `icon_down`
- `title_style`
- `title2_style`
- `text_style`
- `level_style`
- `level_up_style`
- `level_down_style`

7.7.19 media_player

A widget to monitor and contol a media player

Mandatory Arguments

- `entity` - the entity_id of the media player

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `truncate_name` - if specified, the name of the media will be truncated to this length.
- `step` - the step (in percent) that the volume buttons will use. (default, 10%)

Cosmetic Arguments

- `widget_style`
- `icon_on`
- `icon_off`
- `icon_up`
- `icon_down`
- `title_style`
- `title2_style`
- `icon_style_active`
- `icon_style_inactive`
- `text_style`
- `level_style`
- `level_up_style`
- `level_down_style`

7.7.20 group

A widget to monitor and contol a group of lights

Mandatory Arguments

- `entity` - the entity_id of the group

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `monitored_entity` - the actual entity to monitor

Groups currently do no report back state changes correctly when attributes light brightness are changed. As a workaround, instead of looking for state changes in the group, we use `monitored_entity` instead. This is not necessary of there are no dimmable lights in the group, however if there are, it should be set to the entity_id of one of the dimmable group members.

Cosmetic Arguments

- `widget_style`
- `icon_on`
- `icon_off`
- `icon_up`
- `icon_down`

- title_style
- title2_style
- icon_style_active
- icon_style_inactive
- text_style
- level_style
- level_up_style
- level_down_style

7.7.21 navigate

A widget to navgigate to a new URL, intended to be used for switching between dashboards

Mandatory Arguments

Optional Arguments:

- url - a url to navigate to. Use a full URL including the "http" part.
- dashboard - a dashboard to navigate to e.g. MainPanel
- title - the title displayed on the tile
- args - a list of arguments.
- skin - Skin to use with the new screen (for HADash URLs only)

For an arbitary URL, Args can be anything. When specifying a dashboard parameter, args have the following meaning:

timeout - length of time to stay on the new dashboard return - dashboard to return to after the timeout has elapsed.

Both timeout and return must be specified.

If adding arguments use the args variable do not append them to the URL or you may break skinning. Add arguments like this:

```
some_widget:
    widget_type: navigate
    title: Amazon
    url: http://amazon.com
    args:
      arg1: fred
      arg2: jim
```

or:

```
some_widget:
    widget_type: navigate
    title: Sensors
    dashboard: Sensors
    args:
      timeout: 10
      return: Main
```

Cosmetic Arguments

- `icon_active`
- `icon_inactive`
- `widget_style`
- `title_style`
- `title2_style`
- `icon_style`

7.7.22 reload

A widget to reload the current dashboard.

Mandatory Arguments

None.

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text

Cosmetic Arguments

- `icon_active`
- `icon_inactive`
- `widget_style`
- `title_style`
- `title2_style`
- `icon_active_style`
- `icon_inactive_style`

7.7.23 iframe

A widget to display other content within the dashboard

Mandatory Arguments

- `url_list` - a list of 1 or more URLs to cycle though. or
- `img_list` - a list of 1 or more Image URLs to cycle through.

Optional Arguments:

- `title` - the title displayed on the tile
- `refresh` - (seconds) if set, the iframe widget will progress down it's list every refresh period, returning to the beginning when it hits the end. Use this in conjunction with a single entry in the `url_list` to have a single url refresh at a set interval.

For regular HTTP sites, use the `url_list` argument, for images the `img_list` argument should work better.

Example:

```
iframe:
    widget_type: iframe
    title: Cats
    refresh: 60
    url_list:
      - https://www.pexels.com/photo/grey-and-white-short-fur-cat-104827/
      - https://www.pexels.com/photo/eyes-cat-coach-sofa-96938/
      - https://www.pexels.com/photo/silver-tabby-cat-lying-on-brown-wooden-surface-
↪126407/
      - https://www.pexels.com/photo/kitten-cat-rush-lucky-cat-45170/
      - https://www.pexels.com/photo/grey-fur-kitten-127028/
      - https://www.pexels.com/photo/cat-whiskers-kitty-tabby-20787/
      - https://www.pexels.com/photo/cat-sleeping-62640/
```

Content will be shown with scroll bars which can be undesirable. For images this can be alleviated by using an image resizing service such as the one offered by Google.

```
weather_frame:
    widget_type: iframe
    title: Radar
    refresh: 300
    frame_style: ""
    img_list:
      - https://images1-focus-opensocial.googleusercontent.com/gadgets/proxy?
↪url=https://icons.wxug.com/data/weather-maps/radar/united-states/hartford-
↪connecticut-region-current-radar-animation.gif&container=focus&refresh=240&resize_
↪h=640&resize_h=640
      - https://images1-focus-opensocial.googleusercontent.com/gadgets/proxy?
↪url=https://icons.wxug.com/data/weather-maps/radar/united-states/bakersfield-
↪california-region-current-radar.gif&container=focus&refresh=240&resize_h=640&resize_
↪h=640
```

Cosmetic Arguments

- `widget_style`
- `title_style`

7.7.24 camera

A widget to display a refreshing camera image on the dashboard

Mandatory Arguments

- `entity_picture`

This can be found using the developer tools, and will be one of the parameters associated with the camera you want to view. If you are using a password, you will need to append `&api_password=<your password>` to the end of the entity_picture. It will look something like this:

`http://192.168.1.20:8123/api/camera_proxy/camera.living_room?token=<your token>&api_password=<redacted>`

If you are using SSL, remeber to use the full DNS name and not the IP address.

Optional Arguments:

- `refresh` - (seconds) if set, the camera image will refresh every interval.

Cosmetic Arguments

- `widget_style`
- `title_style`

7.7.25 alarm

A widget to report on the state of an alarm and allow code entry

Mandatory Arguments:

- `entity` - the entity_id of the alarm to be monitored

Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text

Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `state_style`
- `panel_state_style`
- `panel_code_style`
- `panel_background_style`
- `panel_button_style`

7.8 Skins

HADashboard fully supports skinning and ships with a number of skins. To access a specific skin, append the parameter `skin=<skin name>` to the dashboard URL. Skin names are sticky if you use the Navigate widet to switch between dashboards and will stay in force until another skin or no skin is specified.

HADasboard currently has the following skins available:

- default - the classic HADashboard skin, very simple
- obsidian, contributed by `@rpitera`
- zen, contributed by `@rpitera`
- simplyred, contributed by `@rpitera`
- glassic, contributed by `@rpitera`

7.9 Skin development

HADashboard fully supports customization through skinning. It ships with a number of skins courtesy of @rpitera, and we encourage users to create new skins and contribute them back to the project.

To create a custom skin you will need to know a little bit of CSS. Start off by creating a directory called `custom_css` in the configuration directory, at the same level as your dashboards directory. Next, create a subdirectory in `custom_css` named for your skin.

The skin itself consists of 2 separate files:

- `dashboard.css` - This is the base dashboard CSS that sets widget styles, background look and feel etc.
- `variables.yaml` - This is a list of variables that describe how different elements of the widgets will look. Using the correct variables you can skin pretty much every element of every widget type.

Dashboard.css is a regular css file, and knowledge of CSS is required to make changes to it.

Variables.yaml is really a set of overrise styles, so you can use fragments of CSS here, basically anything that you could normally put in an HTML `style` tag. Variables .yaml also supports variable expansion to make structuring the file easier. Anything that starts with a `$` is treated as a variable that refers back to one of the other yaml fields in the file.

Here is an example of a piece of a variables.yaml file:

```
##
## Styles
##

white: "#fff"
red: "#ff0055"
green: "#aaff00"
blue: "#00aaff"
purple: "#aa00ff"
yellow: "#ffff00"
orange: "#ffaa00"

gray_dark: "#444"
gray_medium: "#666"
gray_light: "#888"
```

```
##Page and widget defaults
background_style: ""
text_style: ""

##These are used for icons and indicators
style_inactive: "color: $gray_light"
style_active: "color: gold"
style_active_warn: "color: gold"
style_info: "color: gold; font-weight: 500; font-size: 250%"
style_title: "color: gold; font-weight: 900"
style_title2: "color: $white"
```

Here we are setting up some general variables that we can reuse for styling the actual widgets.

Below, we are setting styles for a specific widget, the light widget. All entries are required but can be left blank by using double quotes.

```
light_icon_on: fa-circle
light_icon_off: fa-circle-thin
light_icon_up: fa-plus
light_icon_down: fa-minus
light_title_style: $style_title
light_title2_style: $style_title2
light_icon_style_active: $style_active
light_icon_style_inactive: $style_inactive
light_state_text_style: $white
light_level_style: "color: $gray_light"
light_level_up_style: "color: $gray_light"
light_level_down_style: "color: $gray_light"
light_widget_style: ""
```

Images can be included - create a sub directory in your skin directory, call it `img` or whatever you like, then refer to it in the css as:

`/custom_css/<skin name>/<image directory>/<image filename>`

One final feature is the ability to include additional files in the header and body of the page if required. This can be useful to allow additional CSS from 3rd parties or include JavaScript.

Custom head includes - should be a YAML List inside `variables.yaml`, e.g.:

```
head_includes:
  - some include
  - some other include
```

Text will be included verbatim in the head section of the doc, use for styles, javascript or 3rd party css etc. etc. It is your responsibility to ensure the HTML is correct

Similarly for body includes:

```
body_includes:
  - some include
  - some other include
```

To learn more about complete styles, take a look at the supplied styles to see how they are put together. Start off with the `dashboard.css` and `variables.yaml` from an exisitng file and edit to suit your needs.

7.10 Example Dashboards

Some example dashboards are available in the AppDaemon repository:

Dashboards

CHAPTER 8

HADashboard Widget Development

HADashboard supports a full Widget API intended to simplify the creation of 3rd party widgets. In this guide, we will describe the APIs and requirements for a widget, the workflow for widget creation, and suggestions on how to contribute widgets back to HADashboard.

8.1 What is a Widget?

A widget is a contained piece of functionality that can be placed on a Dashboard. In many cases, widgets refer to types of devices that can be controlled via Home Assistant, but also, widgets can be unrelated, for instance an RSS widget.

There are two main types of widgets, `Base Widgets` and `Derived Widgets`. Base Widgets contain all of the HTML, CSS and JavaScript code to render and run the widget, whereas Derived Widgets are just a structured list of variables that are passed down to Base Widgets. Base Widgets live in subdirectories, Derived Widgets are simply yaml files.

The reason for the 2 types of widget is one of design philosophy. The goal is to have relatively few Base Widgets, and multiple derived widgets that map to them with minor parameter changes. For example, in Home Assistant, a light and a group are fairly similar, and require identical controls and status displays. This makes it possible to create a single Base Widget, and map to it with two separate Derived Widgets. When creating a new Widget type, attempt to do one of the following in order of preference:

1. Create a new Derived Widget that works with an existing Base Widget
2. Create a new Derived Widget that works with modifications to an existing Base Widget
3. Create a new Derived and Base Widget

We also talk abour a third type of widgets, an `Instantiated Widget` - this refers to an actual widget in a dashboard configuration file which will have a widget type and a number of specific variables.

8.2 Creating Custom Widgets

When creating new widgets, in a similar way to custom skins, HADashboard allows the creation of a directory called `custom_widgets` in the configuration directory. Any yaml files placed in here will be treated as new Derived Widgets. Any directories here will be treated as new Base Widgets. If you are creating a new widget you will need to use a new name for the widget. Base Widgets by convention are stored in directories that are named starting with `base` e.g. `baselight`, or `basesuperwidget`.

If either a Derived Widget or Base Widget have the same name as an existing widget, the custom widget will be used in preference to allow existing widgets to be easily modified.

When a widget has been created and tested, and the author desires to contribute the widget back to the community, all that is required is that the Derived and Base Widgets are placed in the Git Repository in the standard widget directory (`appdaemon/widgets`) then a Pull Request may be issued in the ususal way.

8.3 Derived Widgets

A derived widget is simply a `yaml` file with a number of known fields to describe the widget. A secondary function of derived widgets is to map in CSS variables for skinning.

Lets start with an example - here is the derived widget code for the light widget:

```
widget_type: baselight
entity: {{entity}}
post_service_active:
  service: homeassistant/turn_on
  entity_id: {{entity}}
post_service_inactive:
  service: homeassistant/turn_off
  entity_id: {{entity}}
fields:
  title: {{title}}
  title2: {{title2}}
  icon: ""
  units: "%"
  level: ""
  state_text: ""
  icon_style: ""
icons:
  icon_on: $light_icon_on
  icon_off: $light_icon_off
static_icons:
  icon_up: $light_icon_up
  icon_down: $light_icon_down
css:
  icon_style_active: $light_icon_style_active
  icon_style_inactive: $light_icon_style_inactive
static_css:
  title_style: $light_title_style
  title2_style: $light_title2_style
  state_text_style: $light_state_text_style
  level_style: $light_level_style
  unit_style: $light_unit_style
  level_up_style: $light_level_up_style
  level_down_style: $light_level_down_style
  widget_style: $light_widget_style
```

Lets break it down line by line.

8.3.1 Top Level Variables

```
widget_type: baselight
entity: {{entity}}
```

Any entries at the top level are simply variables to be passed to the Base Widget. Some of them have special meanings (listed in the following sections) but any values are allowed, and are all passed to the Base Widget. The exception to this is the `widget_type` entry, which is required and refers to the Base Widget that this Derived Widget works with.

In the example above, `entity` is an argument that will be made available to the base widget. The value, `{{entity}}` is a simple passthrough from the Instantiated Widget in the Dashboard. The significance of this is that a Derived Widget may want to hard code specific parameters while passing others through. For example, a Base Widget may require a `service` parameter for which service to call to turn a device on. A `switch` Derived Widget may hard code this as `switch.turn_on` while a `light` derived widget may hard code it as `light.turn_on`. Both however require the entity name from the Instantiated widget. In practice, this example is somewhat artificial as you could use `home_assistant.turn_on` for both service calls, and in fact lights and switches have different Base Widgets, but the concept remains valid.

An example of the above can be seen in action here:

```
post_service_active:
  service: homeassistant/turn_on
  entity_id: {{entity}}
post_service_inactive:
  service: homeassistant/turn_off
  entity_id: {{entity}}
```

`post_service_active` and `post_service_inactive` are both parameters specific to the baselight Base Widget.

The remaining parameters have special significance and provide required information for the Base Widget.

8.3.2 Fields

```
fields:
  title: {{title}}
  title2: {{title2}}
  icon: ""
  units: "%"
  level: ""
  state_text: ""
  icon_style: ""
```

Entries in the fields arguments map directly to the HTML fields declared in the Base Widget and must all be present. Any field that has a defined value will be used to automatically initialize the corresponding value in the widget. This is useful for static fields such as titles and simplifies the widget code significantly. Fields that are not required to be initialized must still be present and set to `""`. Again, it is possible to map values directly from the Instantiated Widget straight through to the Base Widget.

8.3.3 Icons

```
icons:
  icon_on: $light_icon_on
  icon_off: $light_icon_off
```

The icons parameter refers to icons that may be in use in the Base Widget. The names must match what the Base Widget is expecting. These Icons are expected to be manipulated by the Base Widget and are provided as specific arguments to it. Whilst it is possible to hard code these, the intended use here is to use variables as above. These variables map back to variables in the skin in use and are duplicated, possibly with different values in different skins.

The corresponding skin entries for these in the default skin are:

```
light_icon_on: fa-circle
light_icon_off: fa-circle-thin
```

These could be different in another skin.

In the base widget, there is code to change the icon from the on icon to the off icon in response to a touch or a state change triggered elsewhere. The Base Widget has access to theses icon names when executing that code.

8.3.4 Static Icons

```
static_icons:
  icon_up: $light_icon_up
  icon_down: $light_icon_down
```

Static icons are similar in concept to fields in that they map directly to fields in the widget and will be prepopulated automatically under the assumption that they don't need to change. As with the icons, the actual values are mapped in the skin.

An example of a static icon might be the plus and minus icons on the climate widget - they may be different in other skins but don't need to change once the widget is initialized.

8.3.5 CSS

```
css:
  icon_style_active: $light_icon_style_active
  icon_style_inactive: $light_icon_style_inactive
```

The *css* parameters are analogous to the `icons` - they are styles that are expected to be maipulated as part of the Widget's operation. They will be made available to the widget at initialization time, and are mapped through the skin.

In the case of the light Base Widget they remain the same, but inb a scene for instance, the touch pad is grey except when it is activated when it changes to green - these styles are made available to the Base Widget to use for changing th style when the button is pressed.

8.3.6 Static CSS

```
css:
static_css:
  title_style: $light_title_style
  title2_style: $light_title2_style
```

```
  state_text_style: $light_state_text_style
  level_style: $light_level_style
  unit_style: $light_unit_style
  level_up_style: $light_level_up_style
  level_down_style: $light_level_down_style
  widget_style: $light_widget_style
```

The `statis_css` entry is used for styles that are automatically applied to various fields. As with `static_icons`, these are expected to be static and are automatically applied when the widget initializes. Again, the variables are derived from the skin, and refer top things like titles that remain static for the lifetime of the widget.

8.3.7 Empty Values

None of the special sections `icons`, `static_icons`, `css`, `static_css` can be empty. If no values are required, simply use the yaml syntax for an empty list - `[]`. e.g.:

```
static_icons: []
```

8.3.8 Summary

In summary, a Derived Widget has 2 main functions:

1. Map values from the Instantiated Widget to the Base Widget, supplying hard coded parameters where necessary
2. Interact with the skin in use to provide the correct styles and icons to the Base Widget

It is technically possible to load a Base Widget into a dashboard directly but this is discouraged as it bypasses the skinning. For this reason, even if a Base Widget is used for a single type of widget, a Derived Widget is also required.

8.4 Base Widgets

Base Widgets are where all the work actually gets done. To build a Base Widget you will need an understanding of HTML and CSS as well as proficiency in JavaScript programming. Base Widgets are really just small snippets of HTML code, with associated CSS to control their appearance, and JavaScript to react to touches, and update values based on state changes.

To build a new Base Widget, first create a directory in the appropriate place, named for the widget. By convention, the name of the widget should start with `base` - this is to avoid confusion in the dashboard creation logic between derived and base widgets. The directory will contain 3 files, also named for the widget:

```
hass@Pegasus:/export/hass/src/appdaemon/appdaemon/widgets/baselight$ ls -l
total 16
-rw-rw-r-- 1 hass hass 1312 Mar 19 13:55 baselight.css
-rw-rw-r-- 1 hass hass  809 Mar 19 13:55 baselight.html
-rw-rw-r-- 1 hass hass 6056 Apr 16 10:07 baselight.js
hass@Pegasus:/export/hass/src/appdaemon/appdaemon/widgets/baselight$
```

The files are:

1. An HTML file that describes the various elements that the widget has, such as titles, value fields etc. The HTML file also defines data bindings that the JavaScript piece uses.
2. A CSS File - this describes the basic styles for the widget and is used for placement of elements too

3. A JavaScript file - this file uses the Widget API and contains all of the logic for the widget.

For the pusposes of this document we will provide examples from the `baselight` Base Widget.

8.4.1 Widget HTML Files

The HTML files exist to provide a basic layout for the widget and insert the styles. They are usually fairly simple.

By convention, the variaous tag types have styling suitable for some common elements although that can be overriden in the css file or the skin:

- <h1> is styled for small text such as titles or state text
- <h2> is styled for large icons or text values
- <p> is styled for small unit labels, e.g. `%`

To assist with programatically changing values and styles in the HTML, HADashboard uses Knockout From their web page:

> Knockout is a JavaScript library that helps you to create rich, responsive display and editor user interfaces with a clean underlying data model. Any time you have sections of UI that update dynamically (e.g., changing depending on the user's actions or when an external data source changes), KO can help you implement it more simply and maintainably.

Knockout bindings are used to set various attributes and the binding types in use are as follows:

- data bind - used for setting text values
- attr, type style - used for setting styles
- attr, type class - used for displaying icons

It is suggested that you familiarize yourself with the bindings in use.

Here is an example of an HTML file.

```
<h1 class="title" data-bind="text: title, attr:{style: title_style}"></h1>
<h1 class="title2" data-bind="text: title2, attr:{style: title2_style}"></h1>
<h2 class="icon" data-bind="attr:{style: icon_style}"><i data-bind="attr: {class:␣
↪icon}"></i></h2>
<span class="toggle-area" id="switch"></span>
<p class="state_text" data-bind="text: state_text, attr:{style: state_text_style}"></
↪p>
<div class="levelunit">
<p class="level" data-bind="text: level, attr:{style: level_style}"></p>
<p class="unit" data-bind="html: units, attr:{style: unit_style}"></p>
</div>
<p class="secondary-icon minus"><i data-bind="attr: {class: icon_down, style: level_
↪down_style}" id="level-down"></i></p>
<p class="secondary-icon plus"><i data-bind="attr: {class: icon_up, style: level_up_
↪style}" id="level-up"></i></p>
```

- The first 2 `<h1>` tags set up `title1` and `title2` using a data bind for the values and style attributes to allow the styles to be set. These styles map back to the various `css` and `static_css` supplied as arguments to the widget and their names must match
- The `<h2>` tag introduces a large icon, presumably of a lightbulb or something similar. Here, because of the way that icons work, we are using a class attribute in Knockout to directly set the class of the element which has the effect of forcing an icon to be displayed
- The `<span>` is set up to allow the user to toggle the widget on and off and is reffered to later in the JavaScript

- The here is used for grouping the level and unit labels for the light, along with the included <p> tags which introduce the actual elements
- The last 2 <p> elements are for the up and down icons.

8.4.2 Widget CSS Files

CSS files in widgets are used primarily for positioning of elements since most of the styling occurs in the skins. Since each widget must have a unique id, the ‘‘{id}}‘ piece of each selector name will be substituted with a unique id ewnsuring that even if there are multiple instances of the same widget they will all behave correctly.

Other than that, this is standard CSS used for laying out the various HTML elements appropriately.

Here is an example that works with the HTML above.

```
.widget-baselight-{{id}} {
    position: relative;
}

.widget-baselight-{{id}} .state_text {
    font-size: 85%;
}

.widget-baselight-{{id}} .title {
    position: absolute;
    top: 5px;
    width: 100%;
}

.widget-baselight-{{id}} .title2 {
    position: absolute;
    top: 23px;
    width: 100%;
}

.widget-baselight-{{id}} .state_text {
    position: absolute;
    top: 38px;
    width: 100%;
}

.widget-baselight-{{id}} .icon {
    position: absolute;
    top: 43px;
    width: 100%;
}

.widget-baselight-{{id}} .toggle-area {
    z-index: 10;
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 75%;
}

.widget-baselight-{{id}} .level {
    display: inline-block;
```

```
}

.widget-baselight-{{id}} .unit {
    display: inline-block;
}

.widget-baselight-{{id}} .levelunit {
    position: absolute;
    bottom: 5px;
    width: 100%;
}

.widget-baselight-{{id}} .secondary-icon {
    position: absolute;
    bottom: 0px;
    font-size: 20px;
    width: 32px;
    color: white;
}

.widget-baselight-{{id}} .secondary-icon.plus {
    right: 24px;
}

.widget-baselight-{{id}} .secondary-icon.plus i {
    padding-top: 10px;
    padding-left: 30px;
}

.widget-baselight-{{id}} .secondary-icon.minus {
    left: 8px;
}

.widget-baselight-{{id}} .secondary-icon.minus i {
    padding-top: 10px;
    padding-right: 30px;
}
```

8.4.3 Widget JavaScript Files

The JavaScript file is responsible fore glueing all the pieces together:

- Registering callbacks for events
- Registering callbacks for touches
- Updating the fields, icons, styles as necessary

Lets take a look at a typical JavaScript Widget - the Baselight Widget.

```
function baselight(widget_id, url, skin, parameters)
{
```

All widgets are declared with an initial function named for the widget functions within the .js file although they are technically objects.

This fucntion is in fact the constructor and is initially called when the widget is first loaded. It is handed a number of parameters:

- widget_id - Unique identifier of the widget
- url - the url used to invoke the widget
- the name of the skin in use
- the parameters supplied by the dashboard for this particular widget

Next we need to set up our `self` variable:

```
// Will be using "self" throughout for the various flavors of "this"
// so for consistency ...

self = this
```

For the uninitiated, JavaScript has a somewhat confused notion of scopes when using objects, as scopes can be inherited from different places depebding on the mechanism for calling intto the code. In Widgets, various tricks have been used to present a consistent view to the user which requires an initial declaration of the self variable. From then on, all calls pass this variable between calls to ensure consistency. It is recomended that the convention of declaring `self = this` at the top of the function then rigidly sticking to the use of `self` is adhered to, to avoid confusion.

```
// Initialization

self.widget_id = widget_id

// Parameters may come in useful later on

self.parameters = parameters
```

Here we are storing the parameters in case we need them later.

```
// Parameter handling

if ("monitored_entity" in self.parameters)
{
    entity = self.parameters.monitored_entity
}
else
{
    entity = self.parameters.entity
}

if ("on_brightness" in self.parameters)
{
    self.on_brightness = self.parameters.on_brightness
}
else
{
    self.on_brightness = 127
}
```

Here we process the parameters and set up any variables we may need to refer to later on.

The next step is to set up the widget to respond to various events such as button clicks and state changes.

```
// Define callbacks for on click events
// They are defined as functions below and can be any name as long as the
// 'self'variables match the callbacks array below
// We need to add them into the object for later reference
```

```
self.OnButtonClick = OnButtonClick
self.OnRaiseLevelClick = OnRaiseLevelClick
self.OnLowerLevelClick = OnLowerLevelClick

var callbacks =
    [
        {"selector": '#' + widget_id + ' > span', "action": "click", "callback": self.
↪OnButtonClick},
        {"selector": '#' + widget_id + ' #level-up', "action": "click", "callback":␣
↪self.OnRaiseLevelClick},
        {"selector": '#' + widget_id + ' #level-down', "action": "click", "callback":␣
↪self.OnLowerLevelClick},
    ]
```

Each widget has the opportunity to register itself for button clicks or touches, or any other event type such as `change`. This is done by filling out the callbacks array (which is later used to initialize them). Here we are registering 3 callbacks.

Looking at `OnButtonClick` as an example:

- OnButtonClick is the name of a function we will be declaring later
- self.OnButtonClick is being used to add it to the object
- In Callbacks, we have an entry that connects a jQuery selector to that particular callback, such that when the element identified by the selector is clicked, the callback in the list will be called.
- `action` defines the jQuery action type the callback will respond to, e.g. `click` or `change`

Once the widget is running, the OnButtonClick function will be called whenever the span in the HTML file is touched. You may have noticed that in the CSS file we placed the span on top of everything else and made it cover the entire widget.

Note that there is nothing special about the naming of `OnButtonClick` - it can be called anything as long as the correct references are present in the `callbacks` list.

Next we will setup the state callbacks:

```
// Define callbacks for entities - this model allows a widget to monitor multiple␣
↪entities if needed
// Initial will be called when the dashboard loads and state has been gathered for␣
↪the entity
// Update will be called every time an update occurs for that entity

self.OnStateAvailable = OnStateAvailable
self.OnStateUpdate = OnStateUpdate

var monitored_entities =
    [
        {"entity": entity, "initial": self.OnStateAvailable, "update": self.
↪OnStateUpdate}
    ]
```

This is a similar concept to tracking state changes and displaying them. For the purposes of a widget, we care about 2 separate things:

1. Getting an initial value for the state when the widget is first loaded
2. Tracking changes to the state over time

The first is accomplished by a callback when the widget is first loaded. We add a callback for the entity we are interested in and identify which routine will be called initially whrn the widget is loaded, and which callback will be called whenever we see a state update. These functions will be responsible for updating the fields necessary to show initial state and changes over time. How that happens is a function of the widget design, but for instance a change to a sensor will usually result in that value being displayed in one of the HTML fields.

Here we are tracking just one entity, but it is possible to register callbacks on as many entities as you need for your widget.

When that is in place we finalize the initialization:

```
// Finally, call the parent constructor to get things moving

WidgetBase.call(self, widget_id, url, skin, parameters, monitored_entities, callbacks)
```

After all the setup is complete, we need to make a call to the object's parent constructor to start processing, passing in various parameters, some of which we got from the function call itself, and other like the callbacks that we set up ourselves. The callback parameters must exist but can be empty, e.g. `callbacks = []` - not every widget needs to respond to touches, not every widget needs to respond to state changes.

After this call completes, the initializer is complete and from now on, activity in the widget is governed by callbacks either from initial state, state changes or button clicks,

Next we will define our state callbacks:

```
// Function Definitions

// The StateAvailable function will be called when
// self.state[<entity>] has valid information for the requested entity
// state is the initial state

function OnStateAvailable(self, state)
{
    self.state = state.state;
    if ("brightness" in state.attributes)
    {
        self.level = state.attributes.brightness
    }
    else
    {
        self.level = 0
    }
    set_view(self, self.state, self.level)
}
```

This function was one of the ones that we referred to earlier in the `monitored_entities` list. Since we identified this as the `initial` callback, it will be called with an initial value for the entities state when the widget is first loaded, but after the constructor function has completed. It is handed a self reference, and the state for the entity it subscribed to. What happens when this code is called is up to the widget. In the case of Base Light it will set the icon type depending on whether the light is on or off, and also update the level. Since this is done elsewhere in the widget, I added a function called `set_view` to set these things up. There is also some logic here to account for the fact that in Home Assistant a light has no brightness level if it is off, so `0` is assumed. Here, we also make a note of the current state for later reference - `self.state = state.state`

- `self.state` is an object attribute
- `state.state` is the actual state of the entity. Like othe Home Assistant state descriptions it can also have a set of sub-attributes under `state.attributes` for values like brightness or color etc.

`OnStateUpdate` at least for this widget is very similar to `OnStateAvailable`, in fact it could probably be a single function for both `initial` and `update` but I separated it out for clarity.

```
// The OnStateUpdate function will be called when the specific entity
// receives a state update - it's new values will be available
// in self.state[<entity>] and returned in the state parameter

function OnStateUpdate(self, state)
{
    self.state = state.state;
    if ("brightness" in state.attributes)
    {
        self.level = state.attributes.brightness
    }
    else
    {
        self.level = 0
    }

    set_view(self, self.state, self.level)
}
```

Next we define the functions that we referenced in the `callback` list for the various click actions. First, `OnButtonClick` is responding to someone touching the widget to toggle the state from off to on or vice-versa.

```
function OnButtonClick(self)
{
    if (self.state == "off")
    {
        args = self.parameters.post_service_active
        if ("on_attributes" in self.parameters)
        {
            for (var attr in self.parameters.on_attributes)
            {
                args[attr] = self.parameters.on_attributes[attr]
            }
        }
    }
    else
    {
        args = self.parameters.post_service_inactive
    }
    self.call_service(self, args)
    toggle(self)
}
```

This is less complicated than it looks. What is happening here is that based on the current state of the entity, we are selecting which service to call to change that state. We are looking it up in our parameters that we saved earlier.

So, if the light is `off` we consult our parameters for `post_service_active` whcih should be set to a service that will turn ther light on (e.g. `light/turn_on`). Similarly if it is on, we look for `post_service_inactive` to find out how to turn it off. Once we have made that choice we make the service call to effect the change: `self.call_service()`

The additional logic and loop when state is off is to construct the necessary dictionary of additional parameters in the format the `turn_on` service expects to set brightness, color etc, that may be passed in to the widget.

Raise level is fairly explanatory - this is clicked to make the light brighter:

```
function OnRaiseLevelClick(self)
{
    self.level = self.level + 255/10;
    self.level = parseInt(self.level)
    if (self.level > 255)
    {
        self.level = 255
    }
    args = self.parameters.post_service_active
    args["brightness"] = self.level
    self.call_service(self, args)
}
```

Here we are using `post_service_active` and setting the brightness attribute. Each click will jump 10 units. Lower level is very similar:

```
function OnLowerLevelClick(self)
 {
     self.level = self.level - 255/10;
     if (self.level < 0)
     {
         self.level = 0;
     }
     self.level = parseInt(self.level)
     if (self.level == 0)
     {
         args = self.parameters.post_service_inactive
     }
     else
     {
         args = self.parameters.post_service_active
         args["brightness"] = self.level
     }
     self.call_service(self, args)
 }
```

It is slightly more complex in that rather than setting the level to `0`, when it gets there it turns the light off.

Finally, the toggle function is called by both of the above functions to change the stored state of the entity and update the display (using `set_view()` again)

```
function toggle(self)
{
    if (self.state == "on")
    {
        self.state = "off";
        self.level = 0
    }
    else
    {
        self.state = "on";
    }
    set_view(self, self.state, self.level)
}
```

Set_view() is where we attend to updating the widgets actual display based on the current state that may have just changed.

```
    // Set view is a helper function to set all aspects of the widget to its
    // current state - it is called by widget code when an update occurs
    // or some other event that requires a an update of the view

    function set_view(self, state, level)
    {

        if (state == "on")
        {
            // Set Icon will set the style correctly for an icon
            self.set_icon(self, "icon", self.icons.icon_on)
            // Set view will set the view for the appropriate field
            self.set_field(self, "icon_style", self.css.icon_style_active)
        }
        else
        {
            self.set_icon(self, "icon", self.icons.icon_off)
            self.set_field(self, "icon_style", self.css.icon_style_inactive)
        }
        if (typeof level == 'undefined')
        {
            self.set_field(self, "level", 0)
        }
        else
        {
            self.set_field(self, "level", Math.ceil((level*100/255) / 10) * 10)
        }
    }
}
```

The most important concept here are the 2 calls to update fields:

- set_icon() - update an icon to a different one, usually used to switch from an on representation to an off representation and vice-versa
- set_field() - update a field to show a new value. In this case the brightness field is being update to show the latest value

That is the anatomy of a typical widget - here it is in full:

```
function baselight(widget_id, url, skin, parameters)
{
    // Will be using "self" throughout for the various flavors of "this"
    // so for consistency ...

    self = this

    // Initialization

    self.widget_id = widget_id

    // Parameters may come in useful later on

    self.parameters = parameters

    // Parameter handling

    if ("monitored_entity" in self.parameters)
    {
```

```
        entity = self.parameters.monitored_entity
    }
    else
    {
        entity = self.parameters.entity
    }

    if ("on_brightness" in self.parameters)
    {
        self.on_brightness = self.parameters.on_brightness
    }
    else
    {
        self.on_brightness = 127
    }

    // Define callbacks for on click events
    // They are defined as functions below and can be any name as long as the
    // 'self'variables match the callbacks array below
    // We need to add them into the object for later reference

    self.OnButtonClick = OnButtonClick
    self.OnRaiseLevelClick = OnRaiseLevelClick
    self.OnLowerLevelClick = OnLowerLevelClick

    var callbacks =
        [
            {"selector": '#' + widget_id + ' > span', "callback": self.OnButtonClick},
            {"selector": '#' + widget_id + ' #level-up', "callback": self.
↪OnRaiseLevelClick},
            {"selector": '#' + widget_id + ' #level-down', "callback": self.
↪OnLowerLevelClick},
        ]

    // Define callbacks for entities - this model allows a widget to monitor multiple␣
↪entities if needed
    // Initial will be called when the dashboard loads and state has been gathered␣
↪for the entity
    // Update will be called every time an update occurs for that entity

    self.OnStateAvailable = OnStateAvailable
    self.OnStateUpdate = OnStateUpdate

    var monitored_entities =
        [
            {"entity": entity, "initial": self.OnStateAvailable, "update": self.
↪OnStateUpdate}
        ]

    // Finally, call the parent constructor to get things moving

    WidgetBase.call(self, widget_id, url, skin, parameters, monitored_entities,␣
↪callbacks)

    // Function Definitions

    // The StateAvailable function will be called when
    // self.state[<entity>] has valid information for the requested entity
```

```
    // state is the initial state

    function OnStateAvailable(self, state)
    {
        self.state = state.state;
        if ("brightness" in state.attributes)
        {
            self.level = state.attributes.brightness
        }
        else
        {
            self.level = 0
        }
        set_view(self, self.state, self.level)
    }

    // The OnStateUpdate function will be called when the specific entity
    // receives a state update - it's new values will be available
    // in self.state[<entity>] and returned in the state parameter

    function OnStateUpdate(self, state)
    {
        self.state = state.state;
        if ("brightness" in state.attributes)
        {
            self.level = state.attributes.brightness
        }
        else
        {
            self.level = 0
        }

        set_view(self, self.state, self.level)
    }

    function OnButtonClick(self)
    {
        if (self.state == "off")
        {
            args = self.parameters.post_service_active
            if ("on_attributes" in self.parameters)
            {
                for (var attr in self.parameters.on_attributes)
                {
                    args[attr] = self.parameters.on_attributes[attr]
                }
            }
        }
        else
        {
            args = self.parameters.post_service_inactive
        }
        console.log(args)
        self.call_service(self, args)
        toggle(self)
    }

    function OnRaiseLevelClick(self)
```

```
    {
        self.level = self.level + 255/10;
        self.level = parseInt(self.level)
        if (self.level > 255)
        {
            self.level = 255
        }
        args = self.parameters.post_service_active
        args["brightness"] = self.level
        self.call_service(self, args)
    }

    function OnLowerLevelClick(self)
    {
        self.level = self.level - 255/10;
        if (self.level < 0)
        {
            self.level = 0;
        }
        self.level = parseInt(self.level)
        if (self.level == 0)
        {
            args = self.parameters.post_service_inactive
        }
        else
        {
            args = self.parameters.post_service_active
            args["brightness"] = self.level
        }
        self.call_service(self, args)
    }

    function toggle(self)
    {
        if (self.state == "on")
        {
            self.state = "off";
            self.level = 0
        }
        else
        {
            self.state = "on";
        }
        set_view(self, self.state, self.level)
    }

    // Set view is a helper function to set all aspects of the widget to its
    // current state - it is called by widget code when an update occurs
    // or some other event that requires a an update of the view

    function set_view(self, state, level)
    {

        if (state == "on")
        {
            // Set Icon will set the style correctly for an icon
            self.set_icon(self, "icon", self.icons.icon_on)
            // Set view will set the view for the appropriate field
```

```
            self.set_field(self, "icon_style", self.css.icon_style_active)
        }
        else
        {
            self.set_icon(self, "icon", self.icons.icon_off)
            self.set_field(self, "icon_style", self.css.icon_style_inactive)
        }
        if (typeof level == 'undefined')
        {
            self.set_field(self, "level", 0)
        }
        else
        {
            self.set_field(self, "level", Math.ceil((level*100/255) / 10) * 10)
        }
    }
}
```

8.5 A Note on Skinning

As you have seen, when creating a new wiget, it is also necessary to add entries for the skinning variables. When contributing widgets back, please ensure that you have provided entries for all of the included skins that are sympathetic to the original look and feel, or the PR will not be accepted.

CHAPTER 9

Change Log

9.1 2.1.12 (2017-11-07)

Features

None

Fixes

- Fixed passwords causing 500 error on HADashboard - contributed by wchan.ranelagh

Breaking Changes

None

9.2 2.1.11 (2017-10-25)

Features

None

Fixes

- Fixed an issue with `run_at_sunset()` firing multiple times

Breaking Changes

None

9.3 2.1.10 (2017-10-11)

Features

- Renamed the HADashboard input_slider to input_number to support HASS' change

- Fixed `select_value()` to work with input_number entities

Fixes

None

Breaking Changes

The `input_select` widget has been renamed to `input_number` to support the change in HASS

9.4 2.1.9 (2017-09-08)

Features

None

Fixes

- broken *disable_apps* temporary workaround

Breaking Changes

None

9.5 2.1.8 (2017-09-08)

Features

- Refactor of dashboard code in preparation for HASS integration
- Addition of check to highlight excessive time in scheduler loop
- Split app configuration out into a separate file in preparation for HASS integration
- Enhance widget API to handle all event types instead of just click
- Add example HADashboard focussed Apps for Oslo City Bikes, Caching of local AppDaemon events, Monitoring events and logging, Google Calendar Feed, Oslo Public Transport, YR Weather - contributed by Torkild Retvedt

Fixes

- Fixed a bug that gave a spurious "text widget not found" error

Breaking Changes

- App configuration is now separate from AppDaemon, HASS and HADashboard configuration
- The Widget API has changed to accommodate different event types and now needs an `action` parameter to specify what the event type to be listened for is

9.6 2.1.7 (2017-08-20)

Features

- Converted docs to rst for better readthedocs support
- Added custom widget development
- Enhanced API support to handle multiple endpoints per App

- Added helper functions for Google Home's APP.AI - contributed by engrbm87
- Added `immediate` parameter to listen state to trigger immediate evaluation of the `delay` parameter

Fixes

None

Breaking Changes

- Existing API Apps need to register their endpoint with *register_endpoint()*

9.7 2.1.6 (2017-08-11)

Features

- API now runs on a separate port to the dashboard

Fixes

None

Breaking Changes

- API requires the `api_port` configuration value to be set and now runs on a different port from the dashboard
- SSL Setup for API now requires `api_ssl_certificate` and `api_ssl_key to be set`
- `ad_key` has been renamed to `api_key`

9.8 2.1.5 (2017-08-10)

Features

None

Fixes

None

Breaking Changes

- `get_alexa_slot_value()` now requires a keyword argument for slotname

9.9 2.1.4 (2017-08-10)

Features

None

Fixes

- .cfg file fixes

Breaking Changes

None

9.10 2.1.3 (2017-08-11)

Features

- Restructure docs for readthedocs.io

None

Fixes

None

Breaking Changes

None

9.11 2.1.2 (2017-08-11)

Features

- Add 'get_alexa_slot_value()
- Add `log_size` and `log_generations` config parameters
- Add additional debugging to help Docker users

Fixes

None

Breaking Changes

None

9.12 2.1.0 (2017-08-11)

Features

- Add a reference to official `vkorn` repository for hass.io
- Add the ability to access hass state as App attributes
- Add RESTFul API Support for Apps
- Add `disable_dash` directive to enable API access without Dashboards
- Add Alexa Helper functions
- Update Material Design Icons to 1.9.32 - contributed by minchick
- Use relative URLs for better remote behavior - contributed by Daniel Trnka
- Add SSL Support
- Add Password security for screens and HASS proxying functions
- Add support for secrets in the AppDaemon configuration file
- Add support for secrets in HADashboard configuration files
- `dash_navigate()` now takes an optional screen to return to

Fixes

- Toggle area fixes submitted by azeroth12 and minchick
- Typo fixes submitted by Aaron Linville, vrs01, Gabor SZOLLOSI, Ken Davidson, Christian Lasaczyk, Klaus, Johan Haals
- Fixed missing skin variables for media player and sensor widgets

Breaking Changes

- Compiled dashboards may need to be deleted after this upgrade

9.13 2.0.8 (2017-07-23)

Features

- Add step parameter to media player
- Add `row` parameter to dashboard
- Add ability to set timeout and return on dash navigation
- Add ability to force dashboard page changes from Apps, Alexa and HASS Automations

Fixes

- Add quotes to times in examples.yaml - contributed by Cecron
- Fix python 3.6 issue with datetime.datetime.fromtimestamp() - contributed by motir

Breaking Changes

None

9.14 2.0.7 (2017-07-20)

Features

None

Fixes

- Fixed a bug in label and text_sensor widgets

Breaking Changes

None

9.15 2.0.6 (2017-07-20)

Features

None

Fixes

- Fix a bug causing an apps `terminate()` to not be called

Breaking Changes

None

9.16 2.0.5 (2017-07-16)

Features

None

Fixes

- Change `convert_utc()` to use iso8601 library

Breaking Changes

None

9.17 2.0.4 (2017-07-16)

Features

- AppDaemon is now on PyPi - no more need to use git for installs
- Allow time_zone directive in appdaemon.cfg to override hass supplied time zone
- Add API calls to return info on schedule table and callbacks (get_scheduler_entries(), get_callback_entries())
- Add `get_tracker_details()`
- Add sub entity to sensor
- Add `hass_disconnected` event and allow Apps to run while HASS is disconnected

Fixes

- Fix startup examples to match new `-c` semantics and add in docs
- Fix Time Travel
- Fix for crashes on HASS restart if apps weren't in use - contributed by shprota
- Attempted a fix for `NaN` showing for Nest & Ecobee thermostats when in auto mode

Breaking Changes

None

9.18 2.0.3 (2017-07-09)

Features

- Add error display field to weather widget

Fixes

- Fix issue with device trackers and `use_hass_icon`

Breaking Changes

None

9.19 2.0.2 (2017-07-08)

Features

- Move docker image to python 3.6

Fixes

None

Breaking Changes

None

9.20 2.0.1 (2017-07-08)

Features

- Much Improved Docker support including tutorial - many thanks to quadportnick

Fixes

- Version Change
- Respect cert_path setting when connecting to WebSocket over SSL - contributed by yawor

Breaking Changes

None

9.21 2.0.0beta4 (2017-06-18)

Features

- Migrate timer thread to async
- Add option to turn off verification for self signed certs (contributed by janwh)
- AppDaemon configuration now uses YAML, among other things this allows arbitarily complex nested data structures in App parameters
- Added ability to convert from old cfg file to YAML
- AppDaemon Apps can now publish arbitary state to other Apps and the dashboard
- Added Gauge Widget
- Added RSS Widget
- Add next and previous track to media player

Fixes

- Slider now works correctly after changes outside of HADashboard
- Climate now works correctly after changes outside of HADashboard
- Media player now works correctly after changes outside of HADashboard
- ha.log now correctly dumps data structures
- on_attributes for lights now correctly supports RGB and XY_COLOR

- Fixed a bug in the scheduler to reduce clock skew messages

Breaking Changes

- The cfg file style of configuration is now deprecated although it still works for now for most features
- Argument names passed to Apps are now case sensitive

9.22 2.0.0beta3.5 (2017-04-09)

Features

- Label now accepts HTML for the value
- IFRAME widget now allows vimeo and youtube videos to go fullscreen when clicked
- IFRAME and Camera widgets now have optional title overlay
- Widgets that display icons can now pick up icons defined in HASS
- aiohttp version 2 support

Fixes

-

Breaking Changes

-

9.23 2.0.0beta3 (2017-03-27)

Features

- Added alarm widget
- Added camera widget
- Dimmers and groups now allow you to specify a list of on parameters to control brightness, color etc.
- Edited code for PEP8 Compliance
- Widgets can now have a default size other than `(1x1)`
- Added `empty` to layouts for multiple blank lines
- Numeric values can now have a comma as the decimal separator
- Add Global Parameters
- Rewrote media widget

Fixes

- IFrames now follow widget borders better
- IFrame now allows user input
- Fixed a race condition on dashboard reload

Breaking Changes

- Media Widget now needs to be 2 cells high

9.24 2.0.0beta2 (2017-03-12)

Features

- Widget level styles now correctly override just the styles they are replacing in the skin, not the whole style
- Device tracker toggling of state is optional and defaults to off
- Add climate widget
- Add script widget
- Add lock widget
- Add cover widget
- Added optional `monitored_state` argument to group to pick a representative entity to track dimming instead of guessing
- Introduce new widget definition model in preparation for custom widgets
- Rewrite several widgets using the new model
- Add state map and state text functions to sensor, scene, binary_sensor, switch, device_tracker, script, lock, cover, input_boolean
- Allow dashboard accesses to be logged in a separate file
- Flag to force recompilation after startup
- Additional error checks in many places
- Dashboard determines the stream URL dynamically rather than by having it hard coded
- Add IFRAME widget
- Sensor widget now automatically detects units
- Sensor widget has separate styles for text and numeric
- Style fixes
- Active Map for device trackers

Fixes

- Various minor skin fixes

Breaking Changes

- Widget level styles that relied on overriding the whole skin style may no longer work as expected
- Device trackers must now be explicitly configured to allow the user to toggle state, by setting the `enable` parameter
- Groups of lights must have the `monitored_entity` argument to work properly if they contain any dimmable lights
- `text_sensor` is deprecated and will be removed at some stage. It is now an alias for `sensor`

9.25 2.0.0beta1 (2017-03-04)

Features

- Initial release of HADashboard v2

Fixes

None

Breaking Changes

- appdaemon's `-c` option now identifies a directory not a file. The previously identified file must exist in that directory and be named `appdaemon.cfg`

9.26 1.5.2 (2017-02-04)

Features

- Code formatted to PEP8, various code optimizations - contributed by yawor
- Version check for WebSockets now understands dev versions - contributed by yawor
- `turn_off()` will now call `turn_on()` for scenes since turning a scene off makes no sense, to allow extra flexibility
- Restored the ability to use **line**, **module** and **function** in log messages. Recoded to prevent errors in non-compatible Python versions if the templates are not used.

Fixes

None

Breaking Changes

None

9.27 1.5.1 (2017-01-30)

Features

None

Fixes

- Functionality to substitute line numbers and module names in log statements temporarily removed

Breaking Changes

- Functionality to substitute line numbers and module names in log statements temporarily removed

9.28 1.5.0 (2017-01-21)

Features

- Swap from EventStream to Websockets (Requires Home Assistant 0.34 or later). For earlier versions of HA, AppDaemon will fallback to EventStream.
- Restored less verbose messages on HA restart, but verbose messages can be enabled by setting `-D DEBUG` when starting AppDaemon
- From the command line ctrl-c now results in a clean shutdown.
- Home Assistant config e.g. Latitude, Longitude are now available in Apps in the `self.ha_config` dictionary.

- Logging can now take placeholder strings for line number, function and module which will be appropriately expanded in the actual message
- Add example apps: battery, grandfather, sensor_notification, sound
- Updates to various example apps

Fixes

- get_app() will now return `None` if the app is not found rather than throwing an exception.

Breaking Changes

- get_app() will now return `None` if the app is not found rather than throwing an exception.

None

9.29 1.4.2 (2017-01-21)

Features

None

Fixes

- Remove timeout parameter from SSEClient call unless timeout is explicitly specified in the config file

Breaking Changes

None

9.30 1.4.1 (2017-01-21)

Features

- turn_off() now allows passing of parameters to the underlying service call
- Better handling of scheduler and worker thread errors. More diagnostics, plus scheduler errors now delete the entry where possible to avoid spamming log entries
- More verbose error handling with HA communication errors

Fixes

None

Breaking Changes

None

9.31 1.4.0 (2017-01-20)

Features

- notify() now supports names
- It is now possible to set a timeout value for underlying calls to the HA EventStream
- It is no longer neccesary to specify latitude, longitude and timezone in the config file, the info is pulled from HA

- When being reloaded, Apps are now able to clean up if desired by creating an optional `terminate()` function.
- Added support for module dependencies

Fixes

Breaking Changes

- To include a title when using the `notify()` call, you must now use the keyword `title` instead of the optional positional parameter

9.32 1.3.7 (2017-01-17)

Features

- Add `entity_exists()` call
- List Apps holding up initialization

Fixes

- Add documentation for the days constraint
- Various other contributed documentation fixes

Breaking Changes

None

9.33 1.3.6 (2016-10-01)

Features

- Add device trackers to switch_reset example

Fixes

- Fixed a bug in which AppDaemon exited on startup if HA was not listening causing AppDaemon failure to start on reboots
- Fixed some scheduler behavior for appd and ha restart events
- Fix presence example to only notify when state changes (e.g. not just for position updates)
- Change door notify example to explicitly say "open" or "closed" instead of passing through state
- Fix a bug in device_trackers example

Breaking Changes

None

9.34 1.3.4 (2016-09-20)

Features

- Add Minimote Example
- Add device trackers to switch_reset example

Fixes

- Fixed a minor scheduler bug that didn't honor the delay for callbacks fired from appd and ha restart events

Breaking Changes

None

9.35 1.3.4 (2016-09-18)

Features

- Add Momentary Switch example
- Add Switch Reset Example

Fixes

- Fix a race condition in App Initialization
- Fix a bug that overwrote state attributes
- Fix to smart heat example app
- Fix day constraints while using time travel

Breaking Changes

None

9.36 1.3.3 (2016-09-16)

Features

- Add ability to specify a cert dirctory for self-signed certs
- Add ability for `listen_event()` to listen to any event
- Add filter options to listen_event()

Fixes

- Fix several potential race conditions in the scheduler

Breaking Changes

None

9.37 1.3.2 (2016-09-08)

Features

- Document "Time Travel" functionality
- Add convenience function to set input_select called `select_option()` - contributed by jbardi
- Add global access to configuration and global configuration variables - suggested by ReneTode

Fixes

- Tidy up examples for listen state - suggested by ReneTode

- Warning when setting state for a non-existent entity is now only given the first time
- Allow operation with no `ha_key` specified
- AppDaemon will now use the supplied timezone for all operations rather than just for calculating sunrise and sunset
- Reduce the chance of a spurious Clock Skew error at startup

Breaking Changes

None

9.38 1.3.1 (2016-09-04)

Features

- Add convenience function to set input_selector called `select_value()` - contributed by Dave Banks

Fixes

None

Breaking Changes

None

9.39 1.3.0 (2016-09-04)

Features

- Add ability to randomize times in scheduler
- Add `duration` to listen_state() to fire event when a state condition has been met for a period of time
- Rewrite scheduler to allow time travel (for testing purposes only, no effect on regular usage!)
- Allow input_boolean constraints to have reversed logic
- Add info_listen_state(), info_listen_event() and info_schedule() calls

Fixes

- Thorough proofreading correcting typos and formatting of API.md - contributed by Robin Lauren
- Fixed a bug that was causing scheduled events to fire a second late
- Fixed a bug in `get_app()` that caused it to return a dict instead of an object
- Fixed an error when missing state right after HA restart

Breaking Changes

- `run_at_sunrise()` and `run_at_sunset()` no longer take a fixed offset parameter, it is now a keyword, e.g. `offset = 60`

9.40 1.2.2 (2016-31-09)

Features

None

Fixes

- Fixed a bug preventing get_state() calls for device types
- Fixed a bug that would cause an error in the last minute of an hour or last hour of a day in run_minutely() and run)hourly() respectively

Breaking Changes

None

9.41 1.2.1 (2016-26-09)

Features

- Add support for windows

Fixes

None

Breaking Changes

None

9.42 1.2.0 (2016-24-09)

Features

- Add support for recursive directories - suggested by jbardi

Fixes

None

Breaking Changes

None

9.43 1.1.1 (2016-23-09)

Fixes

- Fix init scripts

9.44 1.1.0 (2016-21-09)

Features

- Installation via pip3 - contributed by Martin Hjelmare
- Docker support (non Raspbian only) - contributed by Jesse Newland
- Allow use of STDERR and SDTOUT as logfile paths to redirect to stdout and stderr respectively - contributed by Jason Hite
- Deprecated "timezone" directive on cfg file in favor of "time_zone" for consistency with Home Assistant config
- Added default paths for config file and apps directory
- Log and error files default to STDOUT and STDERR respectively if not specified
- Added systemd service file - contributed by Jason Hite

Fixes

- Fix to give more information if initial connect to HA fails (but still avoid spamming logs too badly if it restarts)
- Rename 'init' directory to 'scripts'
- Tidy up docs

Breaking Changes

- As a result of the repackaging for PIP3 installation, all apps must be edited to change the import statement of the api to `import appdaemon.appapi as appapi`
- Config must now be explicitly specfied with the -c option if you don't want it to pick a default file location
- Logfile will no longer implicitly redirect to STDOUT if running without the -d flag, instead specify STDOUT in the config file or remove the logfile directive entirely
- timezone is deprecated in favor of time_zone but still works for now

9.45 1.0.0 (2016-08-09)

Initial Release