# AppDaemon Documentation

*Release 4.0.4*

**Andrew Cockburn**

**Jul 11, 2020**

# Contents

AppDaemon is a loosely coupled, multi-threaded, sandboxed python execution environment for writing automation apps for home automation projects, and any environment that requires a robust event driven architecture.

Out of the box, AppDaemon has support for the following automation products:

- Home Assistant home automation software.
- MQTT event broker.

AppDaemon also provides a configurable dashboard (HADashboard) suitable for wall mounted tablets.

AppDaemon is developed and maintained by a small team of hard working folks:

- Andrew Cockburn - AppDaemon founder, Chief Architect and Benevolent Dictator For Life.
- Odianosen Ejale - Core & MQTT Development and maintenance, fixer and tester.
- Humberto Rodríguez Avila - Document Meister and Code Groomer
- Daniel Lashua - Core Developer & JS/CSS/HTML expert and the one that drives them all crazy.

With thanks to previous members of the team:

- Rene Tode
- Robert Pitera

Contents:

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
```

**Do not** install this in the same Python virtual environment as Home Assistant. If you do that, then Home Assistant will stop working.

## 1.3 Install Using hass.io

The official hass.io addon for AppDaemon is maintained by:

- frenck.

## 1.4 Running

### 1.4.1 Docker

Assuming you have set the config up as described in the tutorial 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.

### 1.4.2 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!
```

## 1.5 AppDaemon arguments

```
usage: appdaemon [-h] [-c CONFIG] [-p PIDFILE] [-t TIMEWARP] [-s STARTTIME]
                 [-e ENDTIME]
                 [-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 directory
  -p PIDFILE, --pidfile PIDFILE
                        full path to PID File
  -t TIMEWARP, --timewarp multiplication factor for the scheduler time (see Time␣
↪TraveL)
  -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>
  -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 -e options are for the Time Travel feature and should only be used for testing. They are described in more detail in the API documentation.

## 1.6 Starting At Reboot

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

### 1.6.1 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=%I
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.

### 1.6.2 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.

## 1.7 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.

## 1.8 Windows Support

AppDaemon runs under windows and has been tested with the official 3.8.1 release of python. However, there are a couple of caveats:

- 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.

## 1.9 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 the same way as for Linux distributions, and none of the above Windows Caveats apply to this version. This is the recommended way to run AppDaemon in a Windows 10 and later environment.

## 1.10 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
```

## 1.11 Raspberry Pi Docker

Since the official Docker image isn't compatible with raspberry Pi, you will need to build your own docker image from the downloaded repository.

```
$ git clone https://github.com/home-assistant/appdaemon.git
$ cd appdaemon
```

You can then build and run the docker image locally as follows:

```
$ docker build -t appdaemon --network=host .
$ docker run --name=appdaemon -d -p 5050:5050 \
  --restart=always \
  -e HA_URL="<Your HA_URL value>" \
  -e TOKEN="<your TOKEN value>" \
  -e DASH_URL="http://$HOSTNAME:5050" \
  -v <your_conf_folder>:/conf \
  appdaemon:latest
```

For more information on running AppDaemon under Docker, see the Docker Tutorial. The key difference is that you will be running a locally built instance of AppDaemon rather than one from Docker Hub, so for run commands, make sure to specify "appdaemon:latest" as the image, as above, rather than "acockburn/appdaemon:latest" as the tutorial states.

At the time of writing, @torkildr is maintaining a linked Raspberry Pi image here:

https://hub.docker.com/r/torkildr/rpi-appdaemon/

# Configuration

When you have AppDaemon (AD) 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`.

## 2.1 Initial Setup

Your initial `appdaemon.yaml` file should look something like this if you are using the HASS plugin:

```
appdaemon:
  plugins:
    HASS:
      type: hass
      ha_url: <some_url>
      token: <some_long_lived_access_token>
```

The top level consists of a number of sections:

## 2.2 Log Configuration

The `logs:` section is optional. It consists of a number of log entries that describe the various system and user-defined logs. The logs named `main_log`, `error_log`, `diag_log` and `access_log` have special significance and are used to describe AppDaemons 4 main logs. Any other named log sections will result in the creation of a user-defined log, which can be written to by your apps. The 4 built-in logfiles are used as follows:

- `main_log` is the path to where you want `AppDaemon` to keep its main log.

- `error_log` is the name of the logfile for errors - this will usually be errors during compilation and execution of the apps. If `errorfile = STDERR` errors are sent to stderr instead of a file, if not specified, the output will be sent to STDERR.

- `diag_log` is the name of the log file for diagnostic information. This will contain information form the `log_thread_actions` parameter, as well as information dumped from AppDaemon's internal state when the AppDaemon process is sent a `SIGUSR1` signal.

- `access_log` is the log that AppDaemon will write access information to for HADashboard and the admin interface.

All 4 logs have defaults, so they do not need to be specified, but if any parameters are defined, they will override the defaults.

A simple logs section might look like this:

```
logs:
  main_log:
    filename: /export/pegasus/hass/appdaemon_test/logs/appdaemon.log
  access_log:
    filename: /export/pegasus/hass/appdaemon_test/logs/access.log
  error_log:
    filename: /export/pegasus/hass/appdaemon_test/logs/error.log
  diag_log:
    filename: /export/pegasus/hass/appdaemon_test/logs/diag.log
    log_generations: 5
    log_size: 1024
    format: "{asctime} {levelname:<8} {appname:<10}: {message}"
  test_log:
    name: TestLog
    filename: /export/pegasus/hass/appdaemon_test/logs/test.log
```

All directives are optional with the exception of `name` for user defined logs. The directives have the following meanings:

The directives are as follows:

- `filename` (optional) is the path to where you want the file to be written. If the directive is not specified, the output is sent to STDOUT.

- `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.

- `format` (optional) Format string for the log file - standard str.format() logger format

- `date_format` (optional) - Format string to specify how the date is rendered in standard datetime strftime() format

- `filter_threshold` (optional) - number of repetitions of a log line allowed before filtering starts (default is 1). Setting `filter_threshold` to zero will turn off log filtering entirely - since AppDaemon relies on this mechanism internally to prevent certain types of log loops, this is not recommended.

- `filter_timeout` (optional) - timeout for log filtering. Duplicate log entries that are output less frequently than this value will not have filtering applied (default is 0.9 seconds)

- `filter_repeat_delay` (optional) - when filtering, repeating messages will be printed out periodically every `filter_repeat_delay` seconds (default is 5 seconds)

Note: Filter parameters are set globally, but are applied to individual apps and modules within AppDaemon. This means that log messages interspersed from different apps or from AppDaemon itself even though different, will not reset the filtering of log messages from an individual app if they carry on repeating.

In the above example, a user-defined log called `test_log` has also been created this can be accessed from apps directly from its logger object, or from the self.log call as follows:

```
# Native logger
testlogger = self.get_user_log("test_log")
testlogger.info("Hello %s", "jim")
# self.log()
self.log("Hello", log="test_log")
```

Note that the AppDaemon logs use an enhanced formatter that allows interpolation of the App Name in the logger output as well as all the other standard fields. In addition, the `{asctime}` token will give the right results if time travel is in use. For example, the default logfile format for AppDaemon's main log is:

```
{asctime} {levelname} {appname:<20}: {message}
```

AppDaemon's default time format is `%Y-%m-%d %H:%M:%S.%f%z`.

## 2.3 AppDaemon Configuration

The `appdaemon:` section has a number of directives:

- `filters` (optional) - see below.

- `plugins` (required) - see below.

- `latitude` (required) - latitude for AppDaemon to use (decimal format).

- `longitude` (required) - longitude for AppDaemon to use (decimal format).

- `elevation` (required) - elevation for AppDaemon to use in meters above sea level.

- `time_zone` (required) - timezone for AppDaemon to use (e.g. America/New_York).

- `app_dir` (Optional) - This can be used to place one's apps in a directory, other than under the config directory.

- `exclude_dirs` (optional) - a list of subdirectories to ignore under the apps directory when looking for apps

- `missing_app_warnings` (optional) - by default, AppDaemon will log a warning if it finds a python file that has no associated configuration in an apps.yaml file. If this parameter is set to `1` the warning will be suppressed. This allows non-appdaemon python files to be distributed along with apps.

- `invalid_yaml_warnings` (optional) - by default, AppDaemon will log a warning if it finds an apps.yaml file that doesn't include "class" and "module" for an app. If this parameter is set to `1` the warning will be suppressed. This is intended to ease the distribution of additional yaml files along with apps.

- `production_mode` (optional) - If set to true, AppDaemon will only check for changes in Apps and apps.yaml files when AppDaemon is restarted, as opposed to every second. This can save some processing power on busy systems. Defaults to `False`. This can also be changed from within apps, using the `set_production_mode` API call.

- `thread_duration_warning_threshold` (optional) - AppDaemon monitors the time that each tread spends in an App. If a thread is taking too long to finish a callback, it may impact other apps. AppDaemon will log a warning if any thread is over the duration specified in seconds. The default is 30 seconds, setting this value to `00` will disable the check.

- `log_thread_actions` (optional) - if set to 1, AppDaemon will log all callbacks on entry and exit for the scheduler, events, and state changes - this can be useful for troubleshooting thread starvation issues

When using the `exclude_dirs` directive, you should supply a list of directory names that should be ignored. For example:

```
exclude_dirs:
    - dir1
    - dir2
    - dir3
```

AppDaemon will search for matching directory names at any level of the folder hierarchy under appdir and will exclude that directory and any beneath it. It is not possible to match multiple level directory names e.g., `somedir/dir1`. In that case, the match should be on `dir1`, with the caveat that if you have dir1 anywhere else in the hierarchy, it will also be excluded.

## 2.4 Advanced Appdaemon Configuration

The following items provide a high level of control over AppDaemon's internal functions but for most users should be left at the default settings.

- `total_threads` (optional) - the number of dedicated worker threads to create for running the apps. Normally, AppDaemon will create enough threads to provide one per app, or default to 10 if app pinning is turned off. Setting this to a value will turn off automatic thread management.

- `pin_apps` (optional) - When true (the default) Apps will be pinned to a particular thread which avoids complications around re-entrant code and locking of instance variables

- `pin_threads` (optional) - Number of threads to use for pinned apps, allowing the user to section off a sub-pool just for pinned apps. Default is to use all threads for pinned apps.

- `threadpool_workers` (optional) - the number of max_workers threads to be used by AD internally to execute calls asynchronously. This defaults to `10`.

- `load_distribution` - Algorithm to use for load balancing between unpinned apps. Can be `round-robin` (the default), `random` or `load`

- `timewarp` (optional) - equivalent to the command line flag `-t` but will take precedence

- `qsize_warning_threshold` - total number of items on thread queues before a warning is issued, defaults to 50

- `qsize_warning_step` - when total qsize is over ``qsize_warning_threshold` a warning will be issued every time the `qsize_warning_step` times the utility loop executes (normally once every second), default is 60 meaning the warning will be issued once every 60 seconds.

- `qsize_warning_iterations` - if set to a value greater than 0, when total qsize is over ``qsize_warning_threshold` a warning will be issued every time the `qsize_warning_step` times the utility loop executes but not until the qsize has been excessive for a minimum of `qsize_warning_iterations`. This allows you to tune out brief expected spikes in Q size. Default is 5, usually meaning 5 seconds.

- namespaces (optional) - configure one or more User Defined Namespaces and set their writeback strategy

```
namespaces:
    andrew:
      # writeback is safe, performance or hybrid
      writeback: safe
    jim:
      writeback: performance
    fred:
      writeback: hybrid
```

## 2.5 Secrets

AppDaemon supports the use of *secrets* in the configuration file, 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, or a named file introduced by the top level `secrets:` section. The file should be a simple list of all the secrets. The secrets can be referred to using a !secret value in the configuration file.

The `secret:` section is optional. If it doesn't exist, AppDaemon looks for a file called `secrets.yaml` in the config directory.

An example `secrets.yaml` might look like this:

```
token: ABCDEFG
appdaemon_key: password456
```

The secrets can then be referred to as follows:

```
appdaemon:
  api_key: !secret appdaemon_key
  threads: '10'
  plugins:
    HASS:
      type: hass
      token: !secret token
      ha_url: http://192.168.1.20:8123
```

## 2.6 Filters

The use of filters allows you to run an arbitrary command against a file with a specific extension to generate a new .py file. The use-cases for this are varied, but this can be used to run a preprocessor on an app, or perhaps some kind of global substitute or any of a number of other commands. AppDaemon, when made aware of the filter via configuration, will look for files in the appdir with the specified extension, and run the specified command on them, writing the output to a new file with the specified extension. The output extension would usually be a .py file which would then be picked up by normal app processing, meaning that if you edit the original input file, the result will be a new .py file that is part of an app which will then be restarted.

In addition, it is possible to chain multiple filters, as the filter list is processed in order - just ensure you end with a .py file.

A simple filter would look like this:

```
filters:
  - command_line: /bin/cat $1 > $2
    input_ext: cat
    output_ext: py
```

This would result in AppDaemon looking for any files with the extension `.cat` and running the `/bin/cat` command and creating a file with an extension of `.py`. In the `command_line`, `$1` and `$2` are replaced by the correctly named input and output files. In this example, the output is just a copy of the input, but this technique could be used with commands such as sed and awk, or even m4 for more complex manipulations.

A chained set of filters might look like this:

```
filters:
  - command_line: /bin/cat $1 > $2
```

```
    input_ext: mat
    output_ext: cat
  - command_line: /bin/cat $1 > $2
    input_ext: cat
    output_ext: py
```

These will run in order resulting in edits to a `.mat` file running through the 2 filters and resulting in a new .py file which will run as the app in the usual way.

Finally, it is possible to have multiple unconnected filters like so:

```
filters:
  - command_line: /bin/cat $1 > $2
    input_ext: mat
    output_ext: .py
  - command_line: /bin/cat $1 > $2
    input_ext: cat
    output_ext: py
```

Here we have defined `.mat` and `.cat` files as both creating new apps. In a real-world example the `command_line` would be different.

## 2.7 Plugins

In the example above, you will see that home assistant is configured as a plugin (called HASS). For most applications there is little significance to this - just configure a single plugin for HASS exactly as above. However, for power users, this is a way to allow AppDaemon to work with more than one installation of Home Assistant and/or other plugins such as MQTT.

The plugin architecture also allows the creation of plugins for other purposes, e.g., different home automation systems.

To configure more than one plugin, simply add a new section to the plugins list and configure it appropriately. Before you do this, make sure to review the section on namespaces to fully understand what this entails, and if you are using more than one plugin, make sure you use the namespace directive to create a unique namespace for each plugin. (One of the plugins may be safely allowed to use the default value, however, any more than that will require the namespace directive. There is also no harm in giving them all namespaces, since the default namespace is literally `default` and has no particular significance, it's just a different name, but if you use namespaces other than default you will need to change your Apps to understand which namespaces are in use.).

### 2.7.1 Plugin Configuration

In the required `plugins:` sub-section, there will usually be one or more plugins with a number of directives introduced by a top level name. Some of these are common to all plugins:

- `type` (required) The type of the plugin.

- `namespace` (optional) - which namespace to use. This can safely be left out unless you are planning to use multiple plugins (see below)

- `disable` (optional) - if set to `true`, the plugin will not be loaded - defaults to `false`.

Plugins also support some optional parameters:

- `refresh_delay` - How often the complete state of the plugin is refreshed, in seconds. Default is 600 seconds.

- `refresh_timeout` - How long to wait for the state refresh before cancelling it, in seconds. Default is 30 seconds.

The rest will vary depending upon which plugin type is in use.

## 2.7.2 Configuration of the HASS Plugin

To configure the HASS plugin, in addition to the required parameters above, you will need to add the following:

- `type:` This must be declared and it must be `hass`

- `ha_url` (required for the `hass` plugin) 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 home assistant API password if you have one, otherwise it can be removed. This directive is deprecated - you should use the `token` directive instead

- `token` (required) - set the long-lived token for access to your hass instance (see later for a description of how to create a long-lived access token)

- `cert_verify` (optional) - flag for cert verification for HASS - set to `False` to disable verification on self-signed certs, or certs for which the address used doesn't match the cert address (e.g., using an internal IP address)

- `api_port` (optional) - Port the AppDaemon RESTFul API will listen on. If not specified, the RESTFul API will be turned off.

- `app_init_delay` (optional) - If specified, when AppDaemon connects to HASS each time, it will wait for this number of seconds before initializing apps and listening for events. This is useful for HASS instances that have subsystems that take time to initialize (e.g., zwave).

- `retry_secs` (optional) - If specified, AD will wait for this many seconds in between retries to connect to HASS (default 5 seconds)

- appdaemon_startup_conditions - see *HASS Plugin Startup Conditions*

- plugin_startup_conditions - see *HASS Plugin Startup Conditions*

For example:

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

An example of the HASS plugin could look like the following:

```
secrets: /some/path
log:
  accessfile: /export/hass/appdaemon_test/logs/access.log
  errorfile: /export/hass/appdaemon_test/logs/error.log
  logfile: /export/hass/appdaemon_test/logs/appdaemon.log
  log_generations: 3
  log_size: 1000000
appdaemon:
  threads: 10
  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>
  plugins:
    HASS:
      type: hass
```

(continues on next page)

```
    ha_url: <some_url>
    token: <token>
    cert_path: <path/to/root/CA/cert>
    cert_verify: True
    namespace: default
```

### HASS Authentication

HASS has recently moved to a new authentication model. For programs such as `AppDaemon` it is necessary to create a Long-Lived Access Token, then provide that token to AppDaemon with the `token` directive in the HASS plugin parameters. To create a Long-Lived Access Token for AppDaemon, do the following:

1. Login as the user that you want to create the token for and open the user profile. The profile is found by clicking the icon next to the `Home Assistant` label to the left of the web ui when the burger menu is clicked:
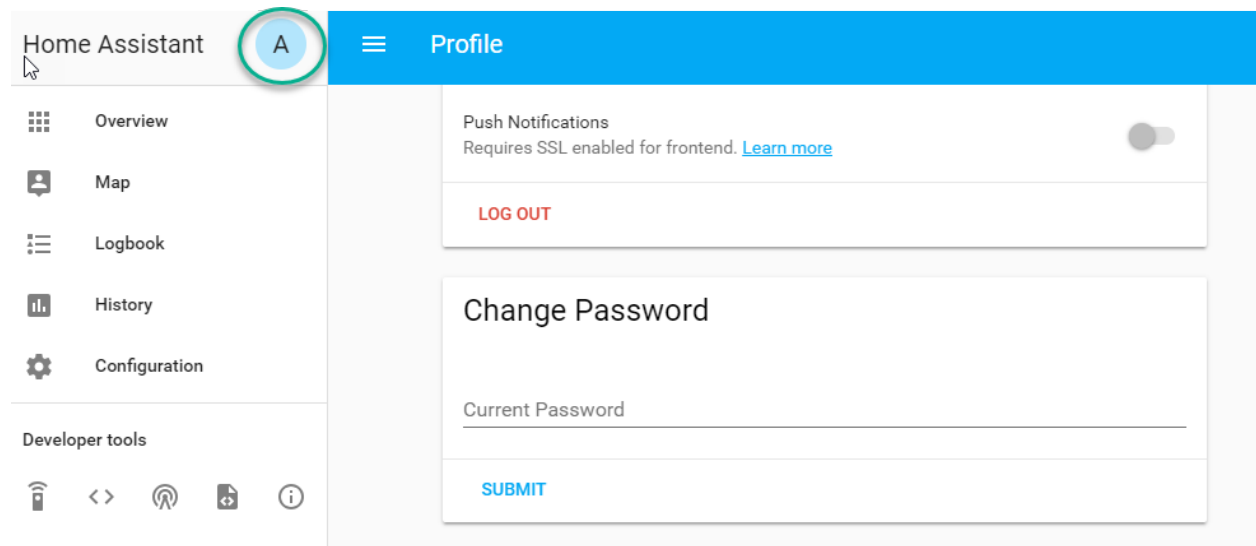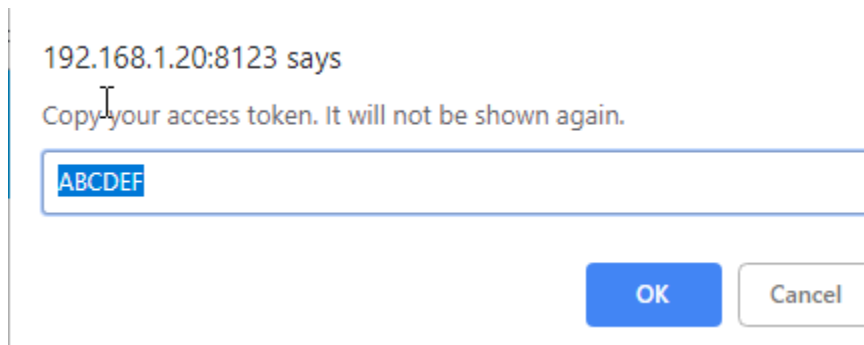


2. At the bottom of the user profile is the Long-Lived Access Tokens section. Click on "Create Token"
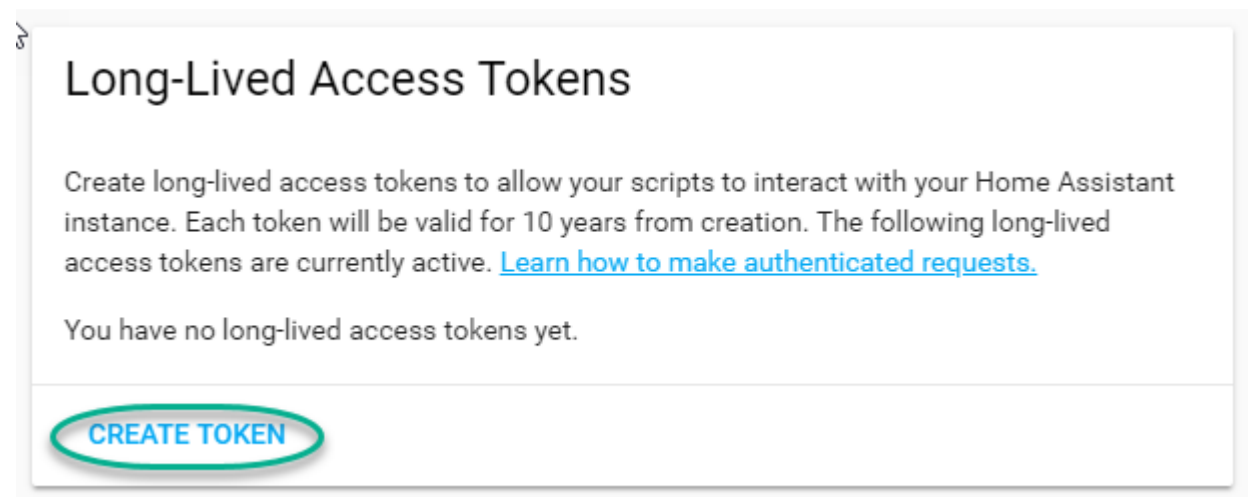


This will pop up a dialog that asks you for the name of the token - this can be anything, it's just to remind you what the token was created for - `AppDaemon` is as good a name as any. When you are done click `OK`

---
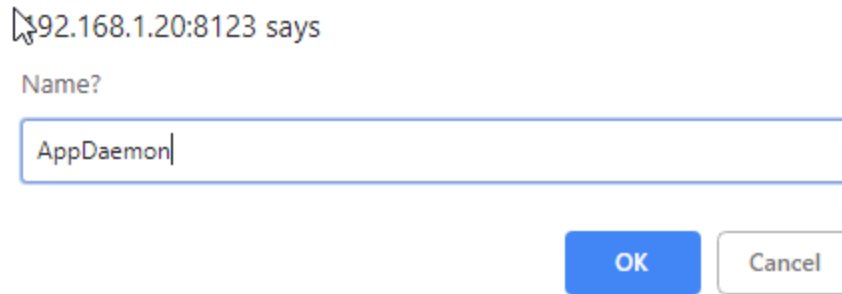
3. A new dialog will popup with the token itself showing:



Copy this string and add it as the argument of the `token` directive in your HASS Plugin section:

```
token: ABCDEF
```

A real token will be a lot longer than this and will consist of a string of random letters and numbers. For example:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiIwZmRkYmE0YTM0MTY0...
```

4. A reference to your new token will be shown in the Long-Lived tokens section, and you can revoke access via this token at any time by pressing the delete icon. The token will last for 10 years.

### HASS Plugin Startup Conditions

The HASS plugin has the ability to pause startup until various criteria have been met. This can be useful to avoid running apps that require certain entities to exist or to wait for an event to happen before the apps are started. There are 2 types of startup criteria, and they are added :

- appdaemon_startup_conditions - conditions that must be met when AppDaemon starts up
- plugin_startup_conditions - conditions that must be met when HASS restarts while AppDaemon is up

AppDamon will pause the startup of the plugin until the conditions have been met. In particular, apps will not have their `initialize()` functions run until the conditions have been met. Each set of conditions takes the same format, and there are 3 types of conditions:

### delay

Delay startup for a number of seconds, e.g.:

```
delay:10
```
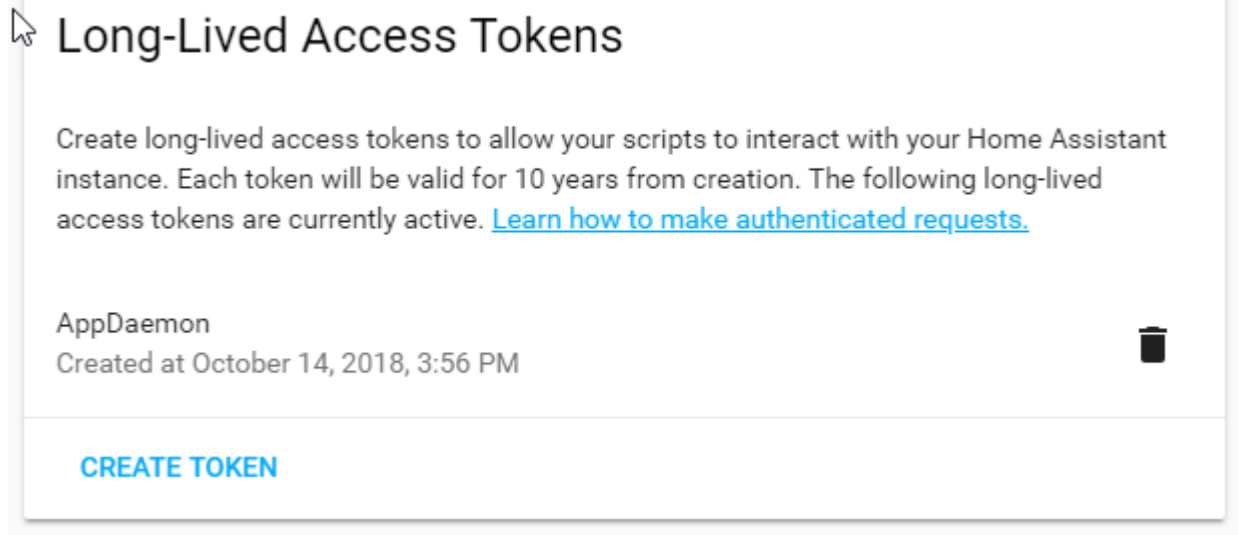
### state

Wait until a specific state exists or has a specific value or set of values. The values are specified as an inline dictionary as follows:

- wait until an entity exists - `state: {entity: <entity id>}`

- wait until an entity exists and has a specific value for its state: `state: {entity: <entity id>, value: {state: "on"}}`

- wait until an entity exists and has a specific value for an attribute: `state: {entity: <entity id>, value: {attributes: {attribute: value}}}`

States and values can be mixed, and they must all match with the state at a point in time for the condition to be satisfied, for instance:

```
state: {entity: light.office_1, value: {state: "on", attributes: {brightness: 254}}}
```

### event

Wait for a specific event.

- wait for a specific event of a given type: `{event_type: <event name>}`

- wait for a specific event with specific data: `{event_type: <event name>, data:{service_data:{entity_id: <some entity>}, service: <some service>}}`

Different condition types may be specified in combination with the following caveats:

- The delay event always executes immediately upon startup, only once. No other checking is performed while the delay is in progress

- State events will be evaluated after any delay every time a new state change event comes in

- Events will be evaluated at the time the event arrives. If there is an additional state event, and it does not match, the event will be discarded, and the plugin will continue to wait until all conditions have been met. This is true even if the state event has previously matched but has reverted to a non-matching state.

**Examples**

Wait for ZWave to complete initialization upon a HASS restart:

```
plugin_startup_conditions:
    event: {event_type: zwave.network_ready}
```

Wait for a specific input boolean to be triggered when AppDaemon restarts:

```
appdaemon_startup_conditions:
    event: {event_type: call_service, data:{domain: homeassistant, service_data:
↪{entity_id: input_boolean.heating}, service: turn_on}}
```

## 2.7.3 Configuration of the MQTT Plugin

To configure the MQTT plugin, in addition to the required parameters above, you will need to add the following:

- `type:` This must be declared and it must be `mqtt`

- `namespace:` (optional) This will default to `default`

- `client_host:` (optional) The IP address or DNS of the Broker. Defaults to 127.0.0.1 which is the localhost

- `client_port:` (optional) The port number used to access the broker. Defaults to `1883`

- `client_transport:` (optional) The transport protocol used to access the broker. This can be either `tcp` or `websockets` Defaults to `tcp`

- `client_clean_session:` (optional) If the broker should clear the data belonging to the client when it disconnects. Defaults to `True`

- `client_id:` (optional) The client id to be used by the plugin, to connect to the broker. If not declared, this will be auto-generated by the plugin. The generated the client id can be retrieved within the app

- `client_user:` (optional) The username to be used by the plugin to connect to the broker. It defaults to `None`, so no username is used

- `client_password:` (optional) The password to be used by the plugin to connect to the broker. It defaults to `None`, so no password is used

- `client_cert:` (optional) The certificate to be used when using SSL

- `tls_version:` (optional) TLS/SSL protocol version to use. Available options are: `auto`, `1.0`, `1.1`, `1.2`. Defaults to `auto`

- `verify_cert:` (optional) This is used to determine if to verify the certificate or not. This defaults to `True` and should be left as True; if not no need having any certificate installed

- `event_name:` (optional) The preferred event name to be used by the plugin. This name is what apps will listen to, to pick up data within apps. This defaults to `MQTT_MESSAGE`

- `client_topics:` (optional) This is a list of topics the plugin is to subscribe to on the broker. This defaults to `#`, meaning it subscribes to all topics on the broker. This can be set to `NONE`, if it is desired to use the subscribe service call within apps, to subscribe to topics.

- `client_qos:` (optional) The quality of service (QOS) level to be used in subscribing to the topics. This will also be used as the default `qos`, when publishing and the qos is not specified by the publishing app.

- `birth_topic:` (optional) This is the topic other clients can subscribe to, to pick up the data sent by the client, when the plugin connects to the broker. If not specified, one is auto-generated

- `birth_payload:` (optional) This is the payload sent by the plugin when it connects to the broker. If not specified, it defaults to `online`

- `birth_retain:` (optional) This tells the broker if it should retain the birth message. If not specified, it defaults to `True`

- `will_topic:` (optional) This is the topic other clients can subscribe to, to pick up the data sent by the broker, when the plugin unceremoniously disconnects from the broker. If not specified, one is auto-generated

- `will_payload:` (optional) This is the payload sent by the broker when the plugin unceremoniously disconnects from the broker. If not specified, it defaults to `offline`

- `will_retain:` (optional) This tells the broker if it should retain the will message. If not specified, it defaults to `True`

- `shutdown_payload:` (optional) This is the payload sent to the broker when the plugin disconnects from the broker cleanly. It uses the same topic as the `will_topic`, and if not specified, defaults to the same payload message and `will_payload`

- `force_start:` (optional) Normally when AD restarts, and the plugin cannot confirm connection to the MQTT broker, it keeps retrying until it has established a connection; this can prevent AD from starting up completely. This can be problematic, if AD is trying to connect to a Cloud broker, and the internet is down. If one is certain of the broker details being correct, and there is a possibility of the broker bring down (e.g., loss of internet connection if using an external broker), the `force_start` flag can be set to `True`. This way AD will start up as usual, and when the broker is online, the plugin will connect to it. This defaults to `False`

All auto-generated data can be picked up within apps, using the `self.get_plugin_config()` API

An example of the MQTT plugin could look like the following:

```
MQTT:
  type: mqtt
  namespace: mqtt
  verbose: True
  client_host: Broker IP Address or DNS
  client_port: Broker PORT Number
  client_id: Client_ID
  client_user: username
  client_password: password
  ca_cert: ca_cert
  tls_version: auto
  client_cert: mycert
  client_key: mykey
  verify_cert: True
  event_name: MQTT_EVENT
  client_topics:
    - hermes/intent/#
    - hermes/hotword/#
```

## 2.8 Configuring a Test App

*This test app assumes the use of the HASS plugin, changes will be required to the app if another plugin is in use.*

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:

```python
import hassapi as hass

#
# Hello World App
#
# Args:
#

class HelloWorld(hass.Hass):

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

Then, we can create a file called apps.yaml in the apps directory and add an entry for the Hello World App like this:

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

App configuration is fully described in the API doc.

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

## 2.9 Configuring the HTTP Component

The HTTP component provides a unified front end to *AppDaemon's Admin Interface*, *HADashboard*, and the *App-Daemon API*. It requires some initial configuration, but the dashboard and admin interface can be separately enabled or disabled. This component also creates a folder in the configuration directory called `web`, if it doesn't exist. To serve custom static content like images, videos or html pages, simply drop the content into the web folder and it becomes available via the browser or dashboard. Content stored in this folder can be accessed using `http://AD_IP:Port/web/<content to be accessed>`. Where *AD_IP:Port* is the url as defined below using the http component.

It has it's own top-level section in AppDaemon.yaml, and one mandatory argument, `url`:

```yaml
http:
    url: http://192.168.1.20:5050
```

- `url` - the URL you want the HTTP component to listen on

To password protect `AppDaemon` use the `password` directive:

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

```yaml
password: !secret ad_password
```

To enable https support for the HTTP Component and by extention the HADashboard and Admin UI, add the following directives pointing to your certificate and keyfile:

AppDaemon uses websockets as the default protocol for streaming events from AppDaemon to the dashboard and admin interface so the dashboard can respond to events in real-time. Some older devices, e.g., original iPad models, do not support websockets. In this case, you may use the alternative socket.io protocol which has better support for older devices. To do this, set the `transport` parameter to `socketio`. The default is `ws` which means the websockets protocol will be used:

```
http:
    transport: socketio
```

Additionally, arbitrary headers can be supplied in all server responses from AppDaemon with this configuration:

```
http:
  headers:
    My-Header-Here: "The Value Of My Header"
```

Headers are especially useful for dealing with CORS. In order to allow CORS from any domain, consider the following configuration:

```
http:
  headers:
    Access-Control-Allow-Origin: "*"
```

This component can also be used to setup custom static directories, which has contents within it that needs to be served using AD's internal web server. This can range from images, videos, html pages and the likes. To do this, consider the configuration below:

```
http:
  custom_dirs:
    videos: /home/pi/video_clips
    pictures: /home/pi/pictures
```

The above configuration assumes that the user has a folder, that has stored within it video clips from like cameras. To access the videos stored in the video_clip folder via a browser or Dashboard, the url can be used `http://AD_IP:Port/videos/<video to be accessed>`. Like wise, the pictures can be accessed using `http://AD_IP:Port/pictures/<picture to be accessed>`.

## 2.10 Configuring the Dashboard

Configuration of the dashboard component (HADashboard) is described separately in the Dashboard documentation. Note that the dashboard depends on the HTTP section being configured to correctly function.

## 2.11 Configuring the API

The AppDaemon App API is configured by adding a top-level directive to appdaemon.yaml:

```
api:
```

It takes no arguments.

## 2.12 Configuring the Admin Interface

The Admin Interface, new in 4.0 is a new front end to AppDaemon that allows you to monitor it's inner workings such as thread activity, registered callbacks and entities. Over time it is expected to evolve into a full management tool for AppDaemon allowing the user to configure, troubleshoot and monitor all of AppDaemon's functions.

The Admin Interface is configured by first adding the HTTP Component and then also adding the top-level directive to appdaemon.yaml:

```
admin:
```

The Interface can be accessed using a web browser and pointing it to the HTTP component URL.

the *admin* directive takes a number of configuration items:

- `title:` The title to be used for the browser window

- `stats_update:` Frequency with which stats are updated in the interface. Allowed values are `none`, `batch`,

`realtime` (default). `none` will turn off updates, `batch` will update the stats every time the utility loop executes, usually every second. `realtime` is recommended for most applications, although if you have a very busy system, operating with sub-second callbacks you may prefer to use `batch` for performance reasons.

## 2.13 Accessing Directories via Apps

Directories used by AD internally either declared by the user or not, can be accessed by the user via apps. The following directories are available:

- `configuration`: self.config_dir

- `apps`: self.app_dir

- `dashboard`: self.dashboard_dir

# 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.

# AppDaemon Tutorial for HASS Users

AppDaemon is a 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.

## 4.1 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 of 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 light switch. However, Home Automation becomes helpful 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 bringing 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 the 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 reuse (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 its 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 AppDaemon 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 its 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 without 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 whatsoever 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 additional functions to make things that were already possible easier.

## 4.2 How it Works

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

### 4.2.1 Sunrise/Sunset Lighting

Let us 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.

```python
import hassapi as hass

class OutsideLights(hass.Hass):

  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.

### 4.2.2 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 callback 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:

```python
import hassapi as hass

class MotionLights(hass.Hass):

  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")
```

At this point, things are starting to get a little more complicated. Home Assistant's automations require an Automation rule and two separate scripts.

Let's use an artificial example to show something that is simple in AppDaemon but very difficult if not impossible using automations. For example, we can 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 also, 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.

```python
import hassapi as hass

class FlashyMotionLights(hass.Hass):

  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 provided 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 you to control when and under what circumstances Apps and callbacks are active to keep the logic clean and simple.

## 4.3 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 several fully fleshed-out examples and tutorials.

**Happy Automating!**

– AppDaemon Team

# AppDaemon with Docker

A quick tutorial to AppDaemon with Docker

## 5.1 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 independent 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!

## 5.2 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!
- You're not running on a Raspberry Pi. See the install page for further information.

## 5.3 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.

- TOKEN: If your Home Assistant is using Long-Lived Tokens you will need to use TOKEN

Now, on your Docker host, for Linux users, run the following command, substituting the values above in the quotes below. (Note: to create a long-lived token, click your user icon in the HA front end and look for the Long-Lived Access Tokens card. If you do not need a TOKEN, you can omit the entire -e TOKEN line)

```
docker run --rm -it -p 5050:5050 \
  -e HA_URL="<your HA_URL value>" \
  -e TOKEN="<your TOKEN 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 outputs:

HASS: Connected to Home Assistant 0.80.0

```
2017-04-01 14:26:48.361140 INFO Connected to Home Assistant 0.80.0
```

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.

```
2018-10-25 16:53:09.105214 INFO Starting Dashboards
```

Now open up a web browser, and browse to http://<DASH_URL>: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...

## 5.4 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 of 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 do something like:

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

Next, we will run a container again, omitting 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 TOKEN="<your TOKEN 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 instructions on AppDaemon configuration. Have fun!

## 5.5 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
```

## 5.6 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 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 TOKEN="<your TOKEN value>" \
  -e DASH_URL="http://$HOSTNAME:5050" \
  -v <your_conf_folder>:/conf \
  acockburn/appdaemon:latest
```

## 5.7 Controlling the AppDaemon Container

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
```

## 5.8 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 \
  acockburn/appdaemon:latest -D DEBUG
```

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

```
docker start appdaemon
```

You can also append any other AppDaemon flags to the end of the command line if desired, e.g. to use time travel.

## 5.9 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 some users:

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

## 5.10 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

## 5.11 Removing AppDaemon

If you no longer want to use AppDaemon `confused`, 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.

## 5.12 Adding Dependencies

Sometimes it can be helpful to install additional Python dependencies into the Docker container before AppDaemon starts, to allow additional libraries to be used from Apps. The Docker script will recursively search the CONF directory for any files named `requirements.txt`. All the found requirements will be used as input to pip3 to install any packages that they describe.

# Writing AppDaemon Apps

AppDaemon (AD) is a loosely coupled, sandboxed, multi-threaded Python execution environment for writing automation apps for Home Assistant, MQTT event broker and other home automation software.

## 6.1 Examples

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

## 6.2 Anatomy of an App

Actions 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 here). This file, is in fact, a Python module, and is expected to contain one or more classes derived from a supplied *AppDaemon class* or a *custom plugin*. For instance, hass support can be used by importing from the supplied `hassapi` module. The start of an App might look like this:

```python
import hassapi as hass

class OutsideLights(hass.Hass):
```

For MQTT you would use the mqttapi module:

```python
import mqttapi as mqtt

class OutsideLights(mqtt.Mqtt):
```

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 a plugin or underlying subsystem such as 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 canceled. 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 three main classes of events:

- Scheduled Events
- State Change Events
- Other 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 App-Daemon'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 HASS App (with comments):

```python
import hassapi as hass
import datetime

# Declare Class
class NightLight(hass.Hass):
  #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)
```

(continues on next page)

```python
    # 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!

## 6.3 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.

## 6.4 Configuration of Apps

Apps are configured by specifying new sections in an app configuration file. The App configuration files exist under the apps directory and can be called anything as long as they end in `.yaml`. You can have one single file for configuration of all apps, or break it down to have one `yaml` file per App, or anything in between. Coupled with the fact that you can have any number of subdirectories for apps and `yaml` files, this gives you the flexibility to structure your apps as you see fit.

The entry for an individual App within a `yaml` file is simply a dictionary entry naming the App, with subfields to supply various parameters. 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 App's 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:

```yaml
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 its 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.

In general, the user should always keep an eye on the error log - system errors will be logged to the main log, any errors that are the responsibility of the user, e.g. that come from app code will be found in the error log.

## 6.5 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

## 6.6 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 first add the apps.yaml entry then the module code and work until it compiles cleanly. 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.

## 6.7 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 arguments, e.g.:

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

Which can be accessed as a list in python with:

```python
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: %
```

It is also possible to get some constants like the app directory within apps. This can be accessed using the attribute `self.app_dir`

### 6.7.1 Secrets

AppDaemon supports the ability to pass sensitive arguments to apps, via the use of secrets in the main or app config file. This will 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, or a named file introduced by the top level `secrets:` section. The file should be a simple list of all the secrets. The secrets can be referred to using a `!secret` tag in the `apps.yaml` file.

An example `secrets.yaml` might look like this:

```
application_api_key: ABCDEFG
```

The secrets can then be referred to in the `apps.yaml` file as follows:

```
appname:
  class: AppClass
  module: appmodule
  application_api_key: !secret application_api_key
```

In the App, the api_key can be accessed like every other argument the App can access.

### 6.7.2 Environment Variables

If not wanting to use the secrets as above, AppDaemon also supports the ability to pass sensitive arguments to apps, via the use of environment variables in the main or app config file. This will allow separate storage of sensitive information such as passwords, within the os's environment variables. The varibales can be referred to using a !env_var tag in the apps.yaml file.

An example using the os's time zone for AD:

```
appdaemon:
  time_zone: !env_var TZ
  latitude: !env_var LAT
  longitude: !env_var LONG
```

The variables can also be referred to in the apps.yaml file as follows:

```
appname:
  class: AppClass
  module: appmodule
  application_api_key: !env_var application_api_key
```

In the App, the api_key can be accessed like every other argument the App can access.

## 6.8 App Dependencies

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

- A global App that defines constants for use in other apps
- An App that provides a service for other modules, e.g., a TTS App

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

AppDaemon fully supports this through the use of the dependency directive in the App configuration. Using this directive, each App identifies other apps that it depends upon. The dependency directive will identify the name of the App it cares about, and AppDaemon will see to it that the dependency is loaded before the App depending on it, and that the dependent App 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

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

It is also possible to have multiple dependencies, added as a yaml list

```
Consumer:
  module: sound
  class: Sound
  dependencies:
    - Sound
    - Global
```

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

Dependencies can also be set using the `register_dependency()` api call.

## 6.9 App Loading Priority

It is possible to influence the loading order of Apps using the dependency system. To add a loading priority to an App, simply add a `priority` entry to its parameters. e.g.:

```
downstairs_motion_light:
  module: motion_light
  class: MotionLight
  sensor: binary_sensor.downstairs_hall
  light: light.downstairs_hall
  priority: 10
```

Priorities can be any number you like, and can be float values if required, the lower the number, the higher the priority. AppDaemon will load any modules with a priority in the order specified.

For modules with no priority specified, the priority is assumed to be `50`. It is, therefore, possible to cause modules to be loaded before and after modules with no priority.

The priority system is complementary to the dependency system, although they are trying to solve different problems. Dependencies should be used when an App literally depends upon another, for instance, it is using variables stored in it with the `get_app()` call. Priorities should be used when an App does some setup for other apps but doesn't provide variables or code for the dependent App. An example of this might be an App that sets up some sensors in Home Assistant, or sets some switch or input_slider to a specific value. It may be necessary for that setup to be performed before other apps are started, but there is no requirement to reload those apps if the first App changes.

To accommodate both systems, dependency trees are assigned priorities in the range 50 - 51, again allowing apps to set priorities such that they will be loaded before or after specific sets of dependent apps.

Note that apps that are dependent upon other apps, and apps that are depended upon by other apps will ignore any priority setting in their configuration.

## 6.10 App Log

Starting from AD 4.0, it is now possible to determine which log as declared by the user, will be used by Apps by default when using the `self.log()` within the App; this can be very useful for debugging purposes. This is done by simply adding the `log:` directive entry, to its parameters. e.g.:

```
downstairs_motion_light:
  module: motion_light
  class: MotionLight
  sensor: binary_sensor.downstairs_hall
  light: light.downstairs_hall
  log: lights_log
```

By declaring the above, each time the function `self.log()` is used within the App, the log entry is sent to the user defined `lights_log`. It is also possible to write to another log, within the same App if need be. This is done using the function `self.log(text, log='main_log')`. Without using any of the aforementioned log capabilities, all logs from apps by default will be sent to the `main_log`.

## 6.11 Global Module Dependencies

The previously described dependencies and load order have all been at the App level. It is however, sometimes convenient to have global modules that have no apps in them that nonetheless require dependency tracking. For instance, a global module might have a number of useful variables in it. When they change, a number of apps may need to be restarted. To configure this dependency tracking, it is first necessary to define which modules are going to be tracked. This is done in any apps.yaml file, although it should only be in one place. We use the `global_modules` directive:

```
global_modules: global
```

This means that the file `globals.py` anywhere with in the apps directory hierarchy is marked as a global module. Any App may simply import `globals` and use its variables and functions. Marking multiple modules as global can be achieved using standard YAML list format:

```
global_modules:
  - global1
  - global2
  - global3
```

Once we have marked the global modules, the next step is to configure any apps that are dependant upon them. This is done by adding a `global_dependencies` field to the App description, e.g.:

```
app1:
  class: App
  module: app
  global_dependencies: global
```

Or for multiple dependencies:

```
app1:
  class: App
  module: app
  global_dependencies:
    - global1
    - global2
```

With this in place, whenever a global module changes that apps depend upon, all dependent apps will be reloaded. This also works well with the App level dependencies. If a change to a global module forces an App to reload that other apps are dependant upon, the dependant apps will also be reloaded in sequence.

## 6.12 Plugin Reloads

When a plugin reloads e.g., due to the underlying system restarting, or a network issue, AppDaemon's default assumption is that all apps could potentially be dependant on that system, and it will force a restart of every App. It is possible to modify this behavior at the individual App level, using the `plugin` parameter in apps.yaml. Specifying a specific plugin or list of plugins will force the App to reload after the named plugin restarts.

For a simple AppDaemon install, the appdaemon.yaml file might look something like this:

```
appdaemon:
  threads: 10
  plugins:
    HASS:
      type: hass
      ha_url: <some_url>
      ha_key: <some_key>
```

In this setup, there is only one plugin, and it is called `HASS` - this will be the case for most AppDaemon users.

To make an App explicitly reload when only this plugin and no other is restarted (e.g., in the case when HASS restarts or when AppDaemon loses connectivity to HASS), use the `plugin` parameter like so:

```
appname:
    module: some_module
    class: some_class
    plugin: HASS
```

If you have more than one plugin, you can make an App dependent on more than one plugin by specifying a YAML list:

```
appname:
    module: some_module
    class: some_class
    plugin:
      - HASS
      - OTHERPLUGIN
```

If you want to prevent the App from reloading at all, just set the `plugin` parameter to some value that doesn't match any plugin name, e.g.:

```
appname:
    module: some_module
    class: some_class
    plugin: NONE
```

Note, that this only effects reloading at plugin restart time:

- apps will be reloaded if the module they use changes

- apps will be reloaded if their apps.yaml changes

- apps will be reloaded when a change to or from DST (Daylight Saving Time) occurs

- apps will be reloaded if an App they depend upon is reloaded as part of a plugin restart

- apps will be reloaded if changes are made to a global module that they depend upon

## 6.13 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. 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, a time callback constraint can be added to an App by adding a parameter to its configuration like this:

```
some_app:
  module: some_module
  class: SomeClass
  constrain_start_time: sunrise
  constrain_end_time: sunset
```

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, unless it is between sunrise and sunset.

An App can have as many or as few constraints 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 be 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.

### 6.13.1 AppDaemon Constraints

AppDaemon itself supplies the time constraint:

#### 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
```

### days

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

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

Other constraints may be supplied by the plugin in use.

## 6.13.2 HASS Plugin Constraints

The HASS plugin supplies several additional different types of constraints:

- input_boolean
- input_select
- presence
- time (see AppDaemon Constraints)

They are described individually below.

### 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
```

**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
```

**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: everyone
# or
constrain_presence: noone
```

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

**person**

The person constraint will constrain based on presence of person entities 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_person: anyone
# or
constrain_person: everyone
# or
constrain_person: noone
```

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

## 6.14 AppDaemon and Threading

AppDaemon is multi-threaded. 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. By default, AppDaemon protects Apps from threading considerations by pinning each App to a specific thread, which means it is not possible for an App to be running in more than one thread at a time. In extremely busy systems this may cause a reduction in performance but this is unlikely.

By default, each App gets its own unique thread to run in. This is generally more threads than are required but it prevents badly behaved apps from blocking other apps pinned to the same thread. This organization can be optimized to use fewer threads if desired by using some of the advanced options below. AppDaemon will dynamically manage the threads for you, creating enough for each App, and adding threads over the lifetime of AppDaemon if new apps are added, to guarantee they all get their own thread.

For most users, threading should be left at the defaults, and things will behave sensibly. If however, you understand concurrency, locking, and re-entrant code, read on for some additional advanced options.

### 6.14.1 Thread Hygiene

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.

### 6.14.2 Disabling App Pinning

If you know what you are doing and understand the risks, you can disable AppDaemon's App Pinning, partially or totally. AppDaemon gives you a huge amount of control, allowing you to enable or disable pinning of individual apps, all apps of a certain class, or even down to the callback level. AppDaemon also lets you explicitly choose which thread apps or callbacks run on, resulting in extremely fine-grained control.

If you disable App pinning, you will start with a default number of 10 threads, but this can be modified with the `total_threads` setting in appdaemon.yaml.

To disable App Pinning globally within AppDaemon set the AppDaemon directive `pin_apps` to `false` within the AppDaemon.yaml file and App pinning will be disabled for all apps. At this point, it is possible for different pieces of code within the App to be executed concurrently, so some care may be necessary if different callbacks, for instance, inspect and change shared variables. This is a fairly standard caveat with concurrent programming, and AppDaemon supplies a simple locking mechanism to help avoid this.

### 6.14.3 Simple Callback Level Locking

The real issue here is that callbacks in an unpinned App can be called at the same time, and even have multiple threads running through them at the same time. To add locking and avoid this, AppDaemon supplies a decorator called `ad.app_lock`. If you use this with any callbacks that manipulate instance variables, you will ensure that there will only be one thread accessing the variables at one time.

Consider the following App which schedules 1000 callbacks all to run at the exact same time, and manipulate the value of `self.important_var`:

```python
import hassapi as hass
import datetime


class Locking(hass.Hass):

    def initialize(self):
        self.important_var = 0
```

```
        now = datetime.datetime.now()
        target = now + datetime.timedelta(seconds=2)
        for i in range (1000):
            self.run_at(self.hass_cb, target)


    def hass_cb(self, kwargs):
        self.important_var += 1
        self.log(self.important_var)
```

As it is, it will result in unexpected results because `self.important_var` can be manipulated by multiple threads at once - for instance, a thread could get the value, add one to it and be just about to write it when another thread jumps in with a different value, which is immediately overwritten. Indeed, when this is run, the output shows just that:

```
2018-11-04 16:07:01.615683 INFO lock: 981
2018-11-04 16:07:01.616150 INFO lock: 982
2018-11-04 16:07:01.616640 INFO lock: 983
2018-11-04 16:07:01.617781 INFO lock: 986
2018-11-04 16:07:01.584471 INFO lock: 914
2018-11-04 16:07:01.621809 INFO lock: 995
2018-11-04 16:07:01.614406 INFO lock: 978
2018-11-04 16:07:01.622616 INFO lock: 997
2018-11-04 16:07:01.619447 INFO lock: 990
2018-11-04 16:07:01.586680 INFO lock: 919
2018-11-04 16:07:01.619926 INFO lock: 991
2018-11-04 16:07:01.620401 INFO lock: 992
2018-11-04 16:07:01.620897 INFO lock: 993
2018-11-04 16:07:01.622156 INFO lock: 996
2018-11-04 16:07:01.603427 INFO lock: 954
2018-11-04 16:07:01.621381 INFO lock: 994
2018-11-04 16:07:01.618622 INFO lock: 988
2018-11-04 16:07:01.623005 INFO lock: 998
2018-11-04 16:07:01.623968 INFO lock: 1000
2018-11-04 16:07:01.623519 INFO lock: 999
```

However, if we add the decorator to the callback function like so:

```
import hassapi as hass
import datetime


class Locking(hass.Hass):

    def initialize(self):
        self.important_var = 0

        now = datetime.datetime.now()
        target = now + datetime.timedelta(seconds=2)
        for i in range (1000):
            self.run_at(self.hass_cb, target)

    @ad.app_lock
    def hass_cb(self, kwargs):
        self.important_var += 1
        self.log(self.important_var)
```

The result is what we would hope for since self.important_var is only being accessed by one thread at a time:

---

```
2018-11-04 16:08:54.545795 INFO lock: 981
2018-11-04 16:08:54.546202 INFO lock: 982
2018-11-04 16:08:54.546567 INFO lock: 983
2018-11-04 16:08:54.546976 INFO lock: 984
2018-11-04 16:08:54.547563 INFO lock: 985
2018-11-04 16:08:54.547938 INFO lock: 986
2018-11-04 16:08:54.548407 INFO lock: 987
2018-11-04 16:08:54.548815 INFO lock: 988
2018-11-04 16:08:54.549306 INFO lock: 989
2018-11-04 16:08:54.549671 INFO lock: 990
2018-11-04 16:08:54.550133 INFO lock: 991
2018-11-04 16:08:54.550476 INFO lock: 992
2018-11-04 16:08:54.550811 INFO lock: 993
2018-11-04 16:08:54.551170 INFO lock: 994
2018-11-04 16:08:54.551684 INFO lock: 995
2018-11-04 16:08:54.552022 INFO lock: 996
2018-11-04 16:08:54.552651 INFO lock: 997
2018-11-04 16:08:54.553033 INFO lock: 998
2018-11-04 16:08:54.553474 INFO lock: 999
2018-11-04 16:08:54.553890 INFO lock: 1000
```

The above scenario is only an issue when thread pinning is disabled. However, another issue with threading arises when apps call each other and modify variables using the `get_app()` call, regardless of whether or not apps are pinned. If a particular App is called at the same time from several different apps using `get_app()`, the App in question will potentially be running on many threads at the same time, and any local resources such as instance variables that are updated could be corrupted. `@ad.app_lock` will also work well to address this situation, if it is applied to the function in the App that is being called. This will force the function to lock using the local lock of the App being called and will enable thread-safe operation.

app1:

```
my_app = get_app("app2")
my_app.myfunction()
```

app2:

```
@ad.app_lock
def my_function()
    self.variable + = 1
```

### 6.14.4 Global Locking

The above style of locking works well for the protection of variables within a single App and across apps using `get_app()`. However, another area where threading might be of concern is if apps are accessing and modifying the dictionary of the global variables which has no locking.

The solution is a global locking decorator called `@ad.global_lock`:

```
@ad.global_lock
def so_something_with_global_vars()
    self.global_vars += 1
```

## 6.14.5 Per-App Pinning

Individual apps can be set to override the global AppDaemon setting for App Pinning by use of the `pin_app` directive in apps.yaml:

```
module: test
class: Test
pin_app: false
```

So if for instance, AppDaemon is set to globally pin apps, the above example will override that and make the App unpinned.

Likewise, if the default is to globally unpin apps, setting `pin_app` to `true` will pin the App.

In addition to controlling pinning, it is also possible to specify the exact thread an App's callbacks will run on, using the `pin_thread` directive:

```
module: test
class: Test
pin_app: true
pin_thread: 6
```

This will result in all callbacks for this App being run by thread 6. The `pin_thread` directive will be ignored if `pin_app` is set to false, or if `pin_app` is not specified and the global setting is to not pin apps.

## 6.14.6 Per Class Pinning

In addition to per-App pinning, it is possible to pin an entire class so that all apps running that code can be pinned or not. This is achieved using an API call, usually in the `initialize()` function that will control whether or not the App is pinned, which will also apply to all apps of the same type since they share the code. Pinning can be enabled or disabled, and thread selected using the pinning API calls:

- `set_app_pin()`
- `get_app_pin()`
- `set_pin_thread()`
- `get_pin_thread()`

These API calls are dynamic, so it is possible to pin and unpin an App as required as well as select the thread it will run on at any point in the Apps lifetime. Callbacks for the scheduler, events or state changes will inherit the values currently set at the time the callback is registered:

```python
# Turn on app pinning
self.set_app_pin(True)
# Select a thread
self.set_pin_thread(5)
# Set a scheduler callback for an hour hence
self.run_in(my_callback, 3600)
# Change the thread
self.set_pin_thread(3)
# Set a scheduler callback for 2 hours hence
self.run_in(my_callback, 7200)
```

The code above will result in 2 callbacks, the first will run on thread 5, the second will run on thread 3.

### 6.14.7 Per Callback Pinning

Per Class Pinning described above, despite its dynamic nature is really intended to be a set and forget setup activity in the apps `initialize()` function. For more dynamic use, it is possible to set the pinning and thread at the callback level, using the `pin` and `pin_thread` parameters to scheduler calls and `listen_state()` and `listen_event()`. These parameters will override the default settings for the App as set in apps.yaml or via the API calls above, but just for the callback in question.

```python
# Turn off app pinning
self.set_app_pin(True)
# Select a thread
self.set_pin_thread(5)
# Set a scheduler callback for an hour hence
self.run_in(my_callback, 3600, pin=False)
```

The above callback will not be pinned.

```python
# Turn off app pinning
set_app_pin(True)
# Select a thread
set_pin_thread(5)
# Set a scheduler callback for an hour hence
run_in(my_callback, 3600, pin_thread=9)
```

The above callback will be run on thread 9, overriding the call to `set_pin_thread()`.

```python
# Set a scheduler callback for an hour hence
run_in(my_callback, 3600, pin=True)
```

The above code is an edge case, if the global or App default is set to not pin. In this case, there won't be an obvious thread to use since it isn't specified, so the callback will default to run on thread 0.

### 6.14.8 Restricting Threads for Pinned Apps

For some usages in mixed pinned and non-pinned environments, it may be desirable to reserve a block of thread specifically for pinned apps. This can be achieved by setting the `pin_threads` directive in AppDamon.yaml:

```
pin_threads: 5
```

In the above example, 5 threads will be reserved for pinned apps, meaning that pinned apps will only run on threads 0 - 4, and will be distributed among them evenly. If the system has 10 threads total, threads 5 - 9 will have no pinned apps running on them, representing spare capacity. In order to utilize the spare threads, you can code apps to explicitly run on them, or set them in the apps.yaml, perhaps reserving threads for specific high priority apps, while the rest of the apps share the lower priority threads. Another way to manage this is via the selection of an appropriate scheduler algorithm.

`pin_threads` will default to the actual number of threads, if App pinning is turned on globally, and it will default to 0 if App pinning is turned off globally. In a mixed setting, if you have any unpinned apps at all you must ensure that `pin_threads` is set to a value less than threads.

### 6.14.9 Scheduler Algorithms

When apps are pinned, there is no choice necessary as to which thread will run a given callback. It will either be selected by AppDaemon, or explicitly specified by the user for each App. For the remainder of unpinned Apps,

AppDaemon must make a choice as to which thread to use, in an attempt to keep the load balanced. There is a choice of 3 strategies, set by the `load_distribution` directive in appdaemon.yaml:

- `roundrobin` (default) - distribute callbacks to threads in a sequential fashion, one thread after another, starting at the beginning when all threads have had their turn. Round Robin scheduling will honor the `pin_threads` directive and only use threads not reserved for pinned apps.

- `random` - distribute callbacks to available threads in a random fashion. Random will also honor the `pin_threads` directive

- `load` - distribute callbacks to the least busy threads (measured by their Q size). Since Load based scheduling is dynamically responding to load, it will take all threads into consideration, including those reserved for pinned apps.

For example:

```
load_distribution: random
```

### 6.14.10 A Final Thought on Threading and Pinning

Although pinning and scheduling has been thoroughly tested, in current real-world applications for AppDaemon, very few of these considerations matter, since in most cases AppDaemon will be able to respond to a callback immediately, and it is unlikely that any significant scheduler queueing will occur unless there are problems with apps blocking threads. At the rate that most people are using AppDaemon, events come in a few times a second, and modern hardware can usually handle the load pretty easily. The considerations above will start to matter more when event rates become a lot faster, by at least an order of magnitude. That is now a possibility with the recent upgrade to the scheduler allowing sub-second tick times, so the ability to lock and pin apps were added in anticipation of new applications for AppDaemon that may require more robust management of apps and much higher event rates.

## 6.15 ASYNC Apps

Note: This is an advanced feature and should only be used if you understand the usage and implications of async programming in Python. If you do not, then the previously described threaded model of apps is much safer and easier to work with.

AppDaemon supports the use of async libraries from within apps as well as allowing a partial or complete async programming model. Callback functions can be converted into coroutines by using the *async* keyword during their declaration. AppDaemon will automatically detect all the App's coroutines and will schedule their execution on the main async loop. This also works for `initialize()` and `terminate()`. Apps can be a mix of *sync* and *async* callbacks as desired. A fully async app might look like this:

```python
import hassapi as hass

class AsyncApp(hass.Hass):

    async def initialize(self):
        # Maybe access an async library to initialize something
        self.run_in(self.hass_cb, 10)

    async def my_function(self):
        # More async stuff here

    async def hass_cb(self, kwargs):
        # do some async stuff
```

```
    # Sleeps are perfectly acceptable
    await self.sleep(10)

    # Call another coroutine
    await my_function()
```

When writing ASYNC apps, please be aware that most of the methods available in ADAPI (generally referenced as `self.method_name()` in an app) are async methods. While these coroutines are automatically turned into a `future` for you, if you intend to use the data they return you'll need to `await` them.

This will not give the expected result:

```
async def some_method(self):
    handle = self.run_in(self.cb, 30)
```

This, however, will:

```
async def some_method(self):
    handle = await self.run_in(self.cb, 30)
```

If you do not need to use the return result of the method, and you do not need to know that it has completed before executing the next line of your code, then you do not need to `await` the method.

### 6.15.1 ASYNC Advantages

- Programming using async constructs can seem natural to advanced users who have used it before, and in some cases, can provide performance benefits depending on the exact nature of the task.

- Some external libraries are designed to be used in an async environment, and prior to AppDaemon async support it was not possible to make use of such libraries.

- Scheduling heavily concurrent tasks is very easy using async

- Using `sleep()` in async apps is not harmful to the overall performance of AppDaemon as it is in regular sync apps

### 6.15.2 ASYNC Caveats

The AppDaemon implementation of ASYNC apps utilizes the same loop as the AppDaemon core. This means that a badly behaved app will not just tie up an individual app; it can potentially tie up all other apps, and the internals of AppDaemon. For this reason, it is recommended that only experienced users create apps with this model.

### 6.15.3 ASYNC Tools

AppDaemon supplies a number of helper functions to make things a little easier:

#### Creating Tasks

For additional multitasking, Apps are fully able to create tasks or futures, however, the app has the responsibility to manage them. In particular, any created tasks or futures must be completed or actively canceled when the app is terminated or reloaded. If this is not the case, the code will not reload correctly due to Pyhton's garbage collection strategy. To assist with this, AppDaemon has a `create_task()` call, which returns a future. Tasks created in

this way can be manipulated as desired, however, AppDaemon keeps track of them and will automatically cancel any outstanding futures if the app terminates or reloads. For this reason, AppDaemon's `create_task()` is the recommended way of doing this.

### Use of Executors

A standard pattern for running I/O intensive tasks such as file or network access in the async programming model is to use executor threads for these types of activities. AppDaemon supplies the `run_in_executor()` function to facilitate this, which uses a predefined thread-pool for execution. As mentioned above, holding up the loop with any blocking activity is harmful not only to the app but all other apps and AppDaemon's internals, so always use an executor for any function that may require it.

### Sleeping

Sleeping in Apps is perfectly fine using the async model. For this purpose, AppDaemon provides the `sleep()` function. If this function is used in a non-async callback, it will raise an exception.

## 6.15.4 ASYNC Threading Considerations

- Bear in mind, that although the async programming model is single threaded, in an event-driven environment such as AppDaemon, concurrency is still possible, whereas in the pinned threading model it is eliminated. This may lead to requirements to lock data structures in async apps.

- By default, AppDaemon creates a thread for each App (unless you are managing the threads yourself). For a fully async app, the thread will be created but never used.

- If you have a 100% async environment, you can prevent the creation of any threads by setting `total_threads:  0` in `appdaemon.yaml`

## 6.16 State Operations

AppDaemon maintains a master state list segmented by namespace. As plugins notify state changes, AppDaemon listens and stores the updated state locally.

The MQTT plugin does not use state at all, and it relies on events to trigger actions, whereas the Home Assistant plugin makes extensive use of state.

## 6.16.1 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 possible 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
```

## 6.16.2 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. Plugins keep track of every state change that occurs within the system, and they streams that information to AppDaemon almost immediately.

A single 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 by 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 users 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,

## 6.16.3 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 scheduling 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 HASS 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 can 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"]))
```

## 6.16.4 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.

## 6.16.5 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.

The kwargs dictionary will also contain a field called `handle` that provides the callback with the handle that identifies the `listen_state()` entry that resulted in the callback.

## 6.17 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 arbitrary state to the dashboard via the 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.

## 6.18 The Scheduler

AppDaemon contains a powerful scheduler that is able to run with microsecond resolution to fire off specific events at set times, or after set delays, or even relative to sunrise and sunset.

### 6.18.1 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.

### 6.18.2 Creation of Scheduler Callbacks

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

### 6.18.3 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 an absolute or relative time or sunrise/sunset depending on which scheduler call you use, and these values affect the base time by the specified 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)
```

## 6.19 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.

## 6.20 Calling Services

### 6.20.1 About Home Assistant 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.

Other plugins may or may not support the notion of services

## 6.21 Events

### 6.21.1 About Events

Events are a fundamental part of how AppDaemon works under the covers. AD receives important events from all of its plugins and communicates them to apps as required. For instance, the MQTT plugin will generate an event when a message is received; The HASS plugin will generate an event when a service is called, or when it starts or stops.

### 6.21.2 Events and MQTT

The MQTT plugin uses events as its primary (and only interface) to MQTT. The model is fairly simple - every time an MQTT message is received, and event of type `MQTT_MESSAGE` is fired. Apps are able to subscribe to this event and process it appropriately.

### 6.21.3 Events and Home Assistant

We have already seen how state changes can be propagated to AppDaemon via the HASS plugin - 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 the HASS plugin, it is possible to subscribe to specific events as well as fire off events.

## 6.21.4 AppDaemon Specific Events

In addition to the HASS and MQTT supplied events, AppDaemon adds 3 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. It is fired within the *global* namespace
- `app_initialized` - fired when an App is initialized. It is fired within the *admin* namespace
- `app_terminated` - fired when an App is terminated. It is fired within the *admin* namespace
- `plugin_started` - fired when a plugin is initialized and properly setup e.g. connection to Home Assistant. It is fired within the plugin's namespace
- `plugin_stopped` - fired when a plugin terminates, or becomes internally unstable like a disconnection from an external system like an MQTT broker. It is fired within the plugin's namespace
- `service_registered` - fired when a service is registered in AD. It is fired within the namespace it was registered
- `stream_connected` - fired when a stream client connects like the Admin User Interface. It is fired within the *admin* namespace
- `stream_disconnected` - fired when a stream client disconnects like the Admin User Interface. It is fired within the *admin* namespace

## 6.21.5 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.

### 6.21.6 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 HASS or MQTT plugin event such as `service_registered` or in the case of HASS, 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)
```

### 6.21.7 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 its 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")
```

### 6.21.8 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

AppDaemon provides convenience functions to assist with this.

### 6.21.9 HASS 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.

### 6.21.10 Writing to Logfiles

AppDaemon uses 2 separate logs - the general log and the error log. An 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 functions are based on the Python `logging` module and are able to pass through parameters for interpolation, and additional parameters such as `exc_info` just as with the usual style of invocation. Use of loggers interpolation method over the use of `format()` is recommended for performance reasons, as logger will only interpolate of the line is actually written whereas `format()` will always do the substitution.

The `-D` option of AppDaemon can be used to specify a global logging level, and Apps can individually have their logging level set as required. This can be achieved using the `set_log_level()` API call, or by using the special `debug` argument to the apps settings in `apps.yaml`:

```
log_level: DEBUG
```

In addition, apps can select a default log for the *log()* call using the *log* directive in apps.yaml, referencing the section name in appdaemon.yaml. This can be one of the 4 builtin logs, `main_log`, `error_log`, `diag_log` and `access_log`, or a user-defined log, e.g.:

```
log: test_log
```

If an App has set a default log other than one of the 4 built in logs, these logs can still be accessed specifically using either the *log=* parameter of the *log()* call, or by getting the appropriate logger object using the *get_user_log()* call, which also works for default logs.

AppDaemon's logging mechanism also allows you to use placeholders for the module, function, 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.

## 6.22 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 - see the section on threading for more details.

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 standard Python nested `Dictionary`.

To get AppDaemon's config parameters for example:

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

To access any apps parameters, use the class attribute called `app_config`. This is a Python Dictionary with an entry for each App, keyed on the App's name.

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

AppDaemon also exposes the configurations from configured plugins. For example, that of the HA plugin allows accessing configurations 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 using the `get_config()` call. E.g.:

```
config = self.get_config()
self.log("My current position is {}(Lat), {}(Long)".format(config["latitude"], config[
→"longitude"]))
```

Using this method, it is also possible to use this function to access configurations of other plugins, from within apps in a different namespace. This is done by simply passing in the `namespace` parameter. E.g.:

And finally, it is also possible to use `config` as a global area for sharing parameters across Apps. Simply add the required parameters inside the appdaemon section in the appdaemon.yaml file:

```
logs:
...
appdaemon:
  global_var: hello world
```

Then access it as follows:

```
my_global_var = self.config["global_var"]
```

## 6.23 Development Workflow

Developing Apps is intended to be fairly simple but is an exercise in programming like any other kind of Python program. 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.

## 6.24 Scheduler Speed

The scheduler has been redesigned in 4.0 with a new tickles algorithm that allows you to specify timed events to the limit of the host system's accuracy (this is usually down to the microsecond level).

## 6.25 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.

### 6.25.1 Choosing a Start Time

Internally, AppDaemon keeps track of its own time relative to when it was started. This make it 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.:

```
$ apprun -s "2018-23-27 16:30:00"
...
2018-12-27 09:31:20.794106 INFO     AppDaemon  App initialization complete
2018-23-27 16:30:00.000000 INFO     AppDaemon  Starting time travel ...
2018-23-27 16:30:00:50.000000 INFO     AppDaemon  Setting clocks to 2018-23-27
→16:30:00
2018-23-27 16:30:00.000000 INFO     AppDaemon  Time displacement factor 1.0
...
```

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

### 6.25.2 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 by speeding uptime. To do this, simply use the -t (timewarp) option on the command line. This option is a simple multiplier for the speed that time will run. If set to 10, time as far as AppDaemon is concerned will run 10 times faster than usual. Set it to 0,1, and time will run 10 times slower. A few examples:

Set appdaemon to run 10x faster than normal:

```
$ appdaemon -t 10
```

Set appdaemon to run as fast as possible:

```
$ appdaemon -t 0
```

The `timewarp` flag in `appdaemon.yaml` is an alternative way of changing the speed, and will override the -t command line setting.

### 6.25.3 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 conjunction 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 10
```

### 6.25.4  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.

### 6.25.5  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.

### 6.25.6  About Plugin Disconnections

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

- When the plugin first goes down or becomes disconnected, an event called `plugin_disconnected` will fire

- While disconnected from the plugin, 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 the plugin will log a warning and return without attempting to contact hass

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

## 6.26  RESTFul API Support

AppDaemon supports a simple RESTFul API to enable arbitrary 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 `api_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 into 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:

```python
import hassapi as hass


class API(hass.Hass):

    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:

```
$ 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:~$
```

## 6.27 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 successful curl example:

```
$ 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:

```
$ curl -i -X POST -H "Content-Type: application/json" http://192.168.1.20:5050/api/
→appdaemon/api -d '{"type": "Hello World Test"}'
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:~$
```

## 6.28 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 you are as most people, you are using SSL to access Home Assistant, there is contention for the 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 its own URL it is possible to have different skills for Home Assistant and AppDaemon.

## 6.29 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 response and card for Alexa.

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

```python
import hassapi as hass
import random
import globals


class Alexa(hass.Hass):

    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("Received 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:
```

(continues on next page)

```python
            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 fahrenheit,".format(self.
→entities.sensor.downstairs_thermostat_temperature.state)
        status += "The upstairs temperature is {} degrees fahrenheit,".format(self.
→entities.sensor.upstairs_thermostat_temperature.state)
        status += "The outside temperature is {} degrees fahrenheit,".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 neighbourhood",
            "Jack is by his bowl waiting for treats"
        ]

        return random.choice(responses)
```

## 6.30 Dialogflow API

Similarly, Dialogflow API for Google home is supported - here is the Google version of the same App. To set up Dialogflow 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.

```python
import hassapi as hass
import random
import globals


class Apiai(hass.Hass):

    def initialize(self):
        pass

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

        if intent is None:
            self.log("Dialogflow 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_dialogflow_response(speech)
            self.log("Received Dialogflow request: {}, answering: {}".format(intent,
→speech))
        else:
            response = self.format_dialogflow_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_dialogflow_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:
```

(continues on next page)

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

    return response

def HouseStatus(self):

    status = "The downstairs temperature is {} degrees fahrenheit,".format(self.
→entities.sensor.downstairs_thermostat_temperature.state)
    status += "The upstairs temperature is {} degrees fahrenheit,".format(self.
→entities.sensor.upstairs_thermostat_temperature.state)
    status += "The outside temperature is {} degrees fahrenheit,".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 neighbourhood",
        "Jack is by his bowl waiting for treats"
    ]

    return random.choice(responses)
```

## 6.31 Plugins

As of version 3.0, AppDaemon has been rewritten to use a pluggable architecture for connection to the systems it monitors.

It is possible to create plugins that interface with other systems, for instance, MQTT support was recently added and it would also be possible to connect to other home automation systems, or anything else for that matter, and expose their operation to AppDaemon and write Apps to monitor and control them.

An interesting caveat of this is that the architecture has been designed so that multiple instances of each plugin can be configured, meaning for instance that it is possible to connect AppDaemon to 2 or more instances of Home Assistant.

To configure additional plugins of any sort, simply add a new section in the list of plugins in the AppDaemon section.

Here is an example of a plugin section with 2 hass instances and 2 dummy instances:

```
plugins:
  HASS1:
    type: hass
    ha_key: !secret home_assistant1_key
    ha_url: http://192.168.1.20:8123
  HASS2:
    namespace: hass2
    type: hass
    ha_key: !secret home_assistant2_key
    ha_url: http://192.168.1.21:8123
  MQTT:
    type: mqtt
    namespace: mqtt
    client_host: 192.168.1.20
    client_port: 1883
    client_id: Fred
    client_user: homeassistant
    client_password: my_password
```

The `type` parameter defines which of the plugins are used, and the parameters for each plugin type will be different. As you can see, the parameters for both hass instances are similar, and it supports all the parameters described in the installation section of the docs - here I am just using a subset.

## 6.32 Namespaces

A critical piece of this is the concept of `namespaces`. Each plugin has an optional `namespace` directive. If you have more than 1 plugin of any type, their state is separated into namespaces, and you need to name those namespaces using the `namespace` parameter. If you don't supply a namespace, the namespace defaults to `default` and this is the default for all areas of AppDaemon meaning that if you only have one plugin you don't need to worry about namespace at all.

In the case above, the first instance had no namespace so its namespace will be called `default`. The second hass namespace will be `hass2` and so on.

These namespaces can be accessed separately by the various API calls to keep things separate, but individual Apps can switch between namespaces at will as well as monitor all namespaces in certain calls like `listen_state()` or `listen_event()` by setting the namespace to `global`.

## 6.32.1 Use of Namespaces in Apps

Each App maintains a current namespace at all times. At initialization, this is set to `default`. This means that if you only have a single plugin, you don't need to worry about namespaces at all as everything will just work.

There are 2 ways to work with namespaces in apps. The first is to make a call to `set_namespace()` whenever you want to change namespaces. For instance, if in the configuration above, you wanted a particular App to work entirely with the `HASS2` plugin instance, all you would need to do is put the following code at the top of your `initialize()` function:

```
self.set_namespace("hass2")
```

Note that you should use the value of the namespace parameter, not the name of the plugin section. From that point on, all state changes, events, service calls, etc. will apply to the `HASS2` instance and the `HASS1` and `DUMMY` instances will be ignored. This is convenient for the case in which you don't need to switch between namespaces.

In addition, most of the API calls allow you to optionally supply a namespace for them to operate under. This will override the namespace set by `set_namespace()` for that call only.

For example:

```
self.set_namespace("hass2")
# Get the entity value from the HASS2 plugin
# Since the HASS2 plugin is configured with a namespace of "hass2"
state = self.get_state("light.light1")

# Get the entity value from the HASS1 plugin
# Since the HASS1 plugin is configured with a namespace of "default"
state = self.get_state("light.light1", namespace="default")
```

In this way it is possible to use a single App to work with multiple namespaces easily and quickly.

## 6.32.2 A Note on Callbacks

One important thing to note, when working with namespaces is that callbacks will honor the namespace they were created with. So if for instance, you create a `listen_state()` callback with a namespace of `default` then later change the namespace to `hass1`, that callback will continue to listen to the `default` namespace.

For instance:

```
self.set_namespace("default")
self.listen_state(callback)
self.set_namespace("hass2")
self.listen_state(callback)
self.set_namespace("dummy1")
```

This will leave us with 2 callbacks, one listening for state changes in `default` and one for state changes in `hass2`, regardless of the final value of the namespace.

Similarly:

```
self.set_namespace("dummy2")
self.listen_state(callback, namespace="default")
self.listen_state(callback, namespace="hass2")
self.set_namespace("dummy1")
```

This code fragment will achieve the same result as above since the namespace is being overridden, and will keep the same value for that callback regardless of what the namespace is set to.

### 6.32.3 User Defined Namespaces

Each plugin has it's own unique namespace as described above, and they are pretty much in control of those namespaces. It is possible to set a state in a plugin managed namespace which can be used as a temporary variable or even as a way of signalling other apps using `listen_state()` however this is not recommended:

- Plugin managed namespaces may be overwritten at any time by the plugin

- They will likely be overwritten when the plugin restarts even if AppDaemon does not

- They will not survive a restart of AppDaemon because it is regarded as the job of the plugin to reconstruct it's state and it knows nothing about any additional variables you have added. Although this technique can still be useful, for example, to add sensors to Home Assistant, a better alternative for Apps to use are User Defined Namespaces.

A User Defined Namespace is a new area of storage for entities that is not managed by a plugin. UDMs are guaranteed not to be changed by any plugin and are available to all apps just the same as a plugin-based namespace. UDMs also survive AppDaemon restarts and crashes, creating durable storage for saving the information and communicating with other apps via `listen_state()` and `set_state()`.

They are configured in the `appdaemon.yaml` file as follows:

```
namespaces:
  my_namespace:
    # writeback is safe, performance or hybrid
    writeback: safe
  my_namespace2:
    writeback: performance
  my_namespace3:
    writeback: hybrid
```

Here we are defining 3 new namespaces - you can have as many as you want. Their names are `my_namespace1`, `my_namespace2` and `my_namespace3`. UDMs are written to disk so that they survive restarts, and this can be done in 3 different ways, set by the writeback parameter for each UDM. They are:

- `safe` - the namespace is written to disk every time a change is made so will be up to date even if a crash happens. The downside is that there is a possible performance impact for systems with slower disks, or that set state on many UDMs at a time.

- `performance` - the namespace is written when AD exits, meaning that all processing is in memory for the best performance. Although this style of UDM will survive a restart, data may be lost if AppDaemon or the host crashes.

- `hybrid` - a compromise setting in which the namespaces are saved periodically (once each time around the utility loop, usually once every second- with this setting a maximum of 1 second of data will be lost if AppDaemon crashes.

## 6.33 Using Multiple APIs From One App

The way apps are constructed, they inherit from a superclass that contains all the methods needed to access a particular plugin. This is convenient as it hides a lot of the complexity by automatically selecting the right configuration information based on namespaces. One drawback of this approach is that an App cannot inherently speak to multiple plugin types as the API required is different, and the App can only choose one API to inherit from.

To get around this, a function called `get_plugin_api()` is provided to instantiate API objects to handle multiple plugins, as a distinct objects, not part of the APPs inheritance. Once the new API object is obtained, you can make plugin-specific API calls on it directly, as well as call `listen_state()` on it to listen for state changes specific to that plugin.

In this case, it is cleaner not to have the App inherit from one or the other specific APIs, and for this reason, the ADBase class is provided to create an App without any specific plugin API. The App will also use `get_ad_api()` to get access to the AppDaemon API for the various scheduler calls.

As an example, this App is built using ADBase, and uses `get_plugin_api()` to access both HASS and MQTT, as well as `get_ad_api()` to access the AppDaemon base functions.

```python
import adbase as ad

class GetAPI(ad.ADBase):

  def initialize(self):

    # Grab an object for the HASS API
    hass = self.get_plugin_api("HASS")
    # Hass API Call
    hass.turn_on("light.office")
    # Listen for state changes for this plugin only
    hass.listen_state(my_callback, "light.kitchen")

    # Grab an object for the MQTT API
    mqtt = self.get_plugin_api("MQTT")
    # Make MQTT API Call
    mqtt.mqtt_publish("topic", "Payload"):

    # Make a scheduler call using the ADBase class
    adbase = self.get_ad_api()
    handle = adbase.run_in(callback, 20)
```

By default, each plugin API object has it's namespace correctly set for that plugin, which makes it much more convenient to handle calls and callbacks form that plugin. This way of working can often be more convenient and clearer than changing namespaces within apps or on the individual calls, so is the recommended way to handle multiple plugins of the same or even different types. The AD base API's namespace defaults to "default":

```python
# Listen for state changes specific to the "HASS" plugin
hass.listen_state(hass_callback, "light.office")
# Listen for state changes specific to the "MQTT" plugin
mqtt.listen_state(mqtt_callback, "light.office")
# Listen for global state changes
adbase.listen_state(global_callback, namespace="global")
```

API objects are fairly lightweight and can be created and discarded at will. There may be a slight performance increase by creating an object for each API in the initialize function and using it throughout the App, but this is likely to be minimal.

## 6.34 Custom Constraints

An App can also register its own custom constraints which can then be used in exactly the same way as App level or callback level constraints. A custom constraint is simply a Python function that returns `True` or `False` when presented with the constraint argument. If it returns `True`, the constraint is regarded as satisfied, and the callback will be made (subject to any other constraints also evaluating to `True`. Likewise, a False return means that the callback won't fire. Custom constraints are a handy way to control multiple callbacks that have some complex logic and enable you to avoid duplicating code in all callbacks.

To use a custom constraint, it is first necessary to register the function to be used to evaluate it using the `register_constraint()` API call. Constraints can also be unregistered using the

deregister_constraint() call, and the list_constraints() call will return a list of currently regis-
tered constraints.

Here is an example of how this all fits together.

We start off with a python function that accepts a value to be evaluated like this:

```python
def is_daylight(self, value):
    if self.sun_up():
        return True
    else:
        return False
```

To use this in a callback level constraint simply use:

```python
self.register_constraint("is_daylight")
handle = self.run_every(self.callback, time, 1, is_daylight=1)
```

Now callback() will only fire if the sun is up.

Using the value parameter you can parameterize the constraint for more complex behavior and use in different situa-
tions for different callbacks. For instance:

```python
def sun(self, value):
    if value == "up":
        if self.sun_up():
        return True
    elif value == "down":
        if self.sun_down():
        return True
    return False
```

You can use this with 2 separate constraints like so:

```python
self.register_constraint("sun")
handle = self.run_every(self.up_callback, time, 1, sun="up")
handle = self.run_every(self.down_callback, time, 1, sun="down")
```

## 6.35 Sequences

AppDaemon supports *sequences* as a simple way of re-using predefined steps of commands. The initial usecase
for sequences is to allow users to create scenes within AppDaemon, however they are useful for many other things.
Sequences are fairly simple and allow the user to define 2 types of activity:

- A call_service command with arbitrary parameters
- A configurable delay between steps.

In the case of a scene, of course you would not want to use the delay, and would just list all the devices to be switched
on or off, however, if you wanted a light to come on for 30 seconds, you could use a script to turn the light on, wait 30
seconds and then turn it off. Unlike in synchronous apps, delays are fine in scripts as they will not hold the apps_thread
up.

There are 2 types of sequence - predefined sequences and inline sequences.

### 6.35.1 Defining a Sequence

A predefined sequence is created by adding a `sequence` section to your apps.yaml file. If you have apps.yaml split into multiple files, you can have sequences defined in each one if desired. For clarity, it is strongly recommended that sequences are created in their own standalone yaml files, ideally in a separate directory from the app argument files.

An example of a simple sequence entry to create a couple of scenes might be:

```yaml
sequence:
  office_on:
    name: Office On
    steps:
    - homeassistant/turn_on:
        entity_id: light.office_1
        brightness: 254
    - homeassistant/turn_on:
        entity_id: light.office_2
        brightness: 254
  office_off:
    name: Office Off
    steps:
    - homeassistant/turn_off:
        entity_id: light.office_1
    - homeassistant/turn_off:
        entity_id: light.office_2
```

The names of the sequences defined above are `sequence.office_on` and `sequence.office_off`. The `name` entry is optional and is used to provide a friendly name for HADashboard. The `steps` entry is simply a list of steps to be taken. They will be processed in the order defined, however without any delays the steps will be processed practically instantaneously.

A sequence to turn a light on then off after a delay might look like this:

```yaml
sequence:
  outside_motion_light:
    name: Outside Motion
    steps:
    - homeassistant/turn_on:
        entity_id: light.outside
        brightness: 254
    - sleep: 30
    - homeassistant/turn_off:
        entity_id: light.outside
```

If you prefer, you can use YAML's inline capabilities for a more compact representation that looks better for longer sequences:

```yaml
sequence:
  outside_motion_light:
    name: Outside Motion
    steps:
    - homeassistant/turn_on: {"entity_id": "light.outside", "brightness": 254}
    - sleep: 30
    - homeassistant/turn_off: {"entity_id": "light.outside"}
```

Sequences can be created that will loop forever by adding the value `loop:    True` to the sequence:

```
sequence:
  outside_motion_light:
    name: Outside Motion
    loop; True
    steps:
    - homeassistant/turn_on: {"entity_id": "light.outside", "brightness": 254}
    - sleep: 30
    - homeassistant/turn_off: {"entity_id": "light.outside"}
```

This sequence once started will loop until either the sequence is canceled, the app is restarted or terminated, or AppDaemon is shutdown.

By default, a sequence will run on entities in the current namespace, however , the namespace can be specified on a per call basis if required.

```
sequence:
  office_on:
    name: Office On
    steps:
    - homeassistant/turn_on:
        entity_id: light.office_1
        brightness: 254
        namespace: "hass1"
    - homeassistant/turn_on:
        entity_id: light.office_2
        brightness: 254
        namespace: "hass2"
```

Just like app parameters and code, sequences will be reloaded after any change has been made allowing scenes to be developed and modified without restarting AppDaemon.

## 6.35.2 Sequence Commands

In addition to a straightforward service name plus data, sequences can take a few additional commands:

- sleep - pause execution of the sequence for a number of seconds. e.g. *sleep: 30* will pause the sequence for 30 seconds
- sequence - run a sub sequence. This must be a predefined sequence, and cannot be an inline sequence. Provide the entity

name of the sub-sequence to be run, e.g. *sequence: sequcene.my_sub_sequence*. Sub sequences can be nested arbitrarily to any desired level.

## 6.35.3 Running a Sequence

Once you have the sequence defined, you can run it in one of 2 ways:

- using the `self.run_sequence()` api call
- Using a sequence widget in HADashboard

A call to run the above sequence would look like this:

```
handle = self.run_sequence("sequence.outside_motion_light")
```

The handle value can be used to terminate a running sequence by supplying it to the `cancel_sequence()` call.

When an app is terminated or reloaded, all running sequences that it started are immediately terminated. There is no way to terminate a sequence started using HADashboard.

### 6.35.4 Inline Sequences

Sequences can be run without the need to predefine them by specifying the steps to the `run_sequence()` command like so:

```
handle = self.run_sequence([
      {'light/turn_on': {'entity_id': 'light.office_1', 'brightness': '5', 'color_
→name': 'white', 'namespace': 'default'}},
      {'sleep': 1},
      {'light/turn_off': {'entity_id': 'light.office_1'}},
      ])
```

# Community Tutorials

Here is a list of other tutorials that have been created by AppDaemon users:

- AppDaemon For Beginners
- AppDaemon Tutorial #1 Tracker-Notifier
- AppDaemon Tutorial #2 Errorlog Notifications
- AppDaemon Tutorial #3 Utility Functions
- AppDaemon Tutorial #4 Libraries & Interactivity
- Home Presence Appdaemon App
- App #1: Doorbell notification
- App #2: Smart Light
- App #3: Smart Radiator
- App #4: Boiler Alert
- App #5: Smart Radiator (Generic)
- App #6: Window Alert
- App #7: Boiler Temperature Alert
- App #8: Detect a particular sequence of events

Do you have other tutorials? Make a PR :)

– AppDaemon Team

# AppDaemon API Reference

A number of api calls are native to AppDaemon and will exist in any App as they are inherited through the plugin API. If the `get_plugin_api()` style of declarations is used, these functions will become available via an object created by the `get_ad_api()` call:

```python
import adbase as ad
import adapi as adapi


class Test(ad.ADBase):

  def initialize(self):

    adbase = self.get_ad_api()
    handle = self.adbase.run_in(callback, 20)
```

These calls are documented below.

## 8.1 App Creation

To create apps based on just the AppDaemon base API, use some code like the following:

```python
import adbase as ad

class MyApp(ad.ADBase):

  def initialize(self):
```

## 8.2 Reference

### 8.2.1 State

**get_state**(*self*, *entity_id=None*, *attribute=None*, *default=None*, *copy=True*, *\*\*kwargs*)
Gets 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. In other words, states are updated using a push-based approach instead of a pull-based one.

> **Parameters**
>
> - **entity_id** (*str, optional*) – 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** (*str, optional*) – 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. 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.
>
> - **default** (*any, optional*) – The value to return when the requested attribute or the whole entity doesn't exist (Default: `None`).
>
> - **copy** (*bool, optional*) – By default, a copy of the stored state object is returned. When you set `copy` to `False`, you get the same object as is stored internally by AppDaemon. Avoiding the copying brings a small performance gain, but also gives you write-access to the internal AppDaemon data structures, which is dangerous. Only disable copying when you can guarantee not to modify the returned state object, e.g., you do read-only operations.
>
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.
>
> **Returns** The entire state of Home Assistant at that given time, if if `get_state()` is called with no parameters. This will consist of a dictionary with a key for each entity. Under that key will be the standard entity state information.

#### Examples

Get the state of the entire system.

```
>>> state = self.get_state()
```

Get the state of all switches in the system.

```
>>> state = self.get_state("switch")
```

Get the state attribute of *light.office_1*.

```
>>> state = self.get_state("light.office_1")
```

Get the brightness attribute of *light.office_1*.

```
>>> state = self.get_state("light.office_1", attribute="brightness")
```

Get the entire state of *light.office_1*.

```
>>> state = self.get_state("light.office_1", attribute="all")
```

**set_state**(*self*, *entity*, *\*\*kwargs*)
   Updates the state of the specified entity.

   **Parameters**

   - **entity** (*str*) – The fully qualified entity id (including the device type).

   - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

   **Keyword Arguments**

   - **state** – New state value to be set.

   - **attributes** (*optional*) – Entity's attributes to be updated.

   - **namespace** (*str, optional*) – If a *namespace* is provided, AppDaemon will change the state of the given entity in the given namespace. On the other hand, if no namespace is given, AppDaemon will use the last specified namespace or the default namespace. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.

   - **replace** (*bool, optional*) – If a *replace* flag is given and set to `True` and `attributes` is provided, AD will attempt to replace its internal entity register with the newly supplied attributes completely. This can be used to replace attributes in an entity which are no longer needed. Do take note this is only possible for internal entity state. For plugin based entities, this is not recommended, as the plugin will mostly replace the new values, when next it updates.

   **Returns** A dictionary that represents the new state of the updated entity.

   **Examples**

   Update the state of an entity.

```
>>> self.set_state("light.office_1", state="off")
```

   Update the state and attribute of an entity.

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

   Update the state of an entity within the specified namespace.

```
>>> self.set_state("light.office_1", state="off", namespace ="hass")
```

**listen_state**(*self*, *callback*, *entity=None*, *\*\*kwargs*)
   Registers a callback to react to state changes.

   This function allows the user to register a callback for a wide variety of state changes.

   **Parameters**

   - **callback** – Function to be invoked when the requested state change occurs. It must conform to the standard State Callback format documented here

- **entity** (`str, optional`) – 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.

- **\*\*kwargs** (`optional`) – Zero or more keyword arguments.

**Keyword Arguments**

- **attribute** (`str, 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`.

- **duration** (`int, 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 requires that a specific attribute is specified (or the default of `state` is used), and should be used 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 occurred, since the assumption is that none of them have changed in the intervening period.

  If you use `duration` when listening for an entire device type rather than a specific entity, or for all state changes, you may get unpredictable results, so it is recommended that this parameter is only used in conjunction with the state of specific entities.

- **timeout** (`int, optional`) – If `timeout` is supplied as a parameter, the callback will be created as normal, but after `timeout` seconds, the callback will be removed. If activity for the listened state has occurred that would trigger a duration timer, the duration timer will still be fired even though the callback has been deleted.

- **immediate** (`bool, optional`) – It enables the countdown for a delay parameter to start at the time, if given. If the `duration` parameter is not given, the callback runs immediately. What this means is that after the callback is registered, rather than requiring one or more state changes before it runs, it immediately checks the entity's states based on given parameters. If the conditions are right, the callback runs immediately at the time of registering. This can be useful if, for instance, you want the callback to be triggered immediately if a light is already *on*, or after a `duration` if given.

  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. If `new` and `duration` are not set, `immediate` will trigger the callback immediately and report in its callback the new parameter as the present state of the entity. If `attribute` is specified, the state of the attribute will be used instead of state. In these cases, `old` will be ignored and when the callback is triggered, its state will be set to `None`.

- **oneshot** (`bool, optional`) – If `True`, the callback will be automatically cancelled after the first state change that results in a callback.

- **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter. The value `global` for namespace has special significance and means that the callback will listen to state updates from any plugin.

- **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.

- **pin_thread** (*int, optional*) – Sets which thread from the worker pool the callback will be run by (0 - number of threads -1).

- **\*kwargs** (*optional*) – Zero or more keyword arguments that will be supplied to the callback when it is called.

### Notes

The `old` and `new` args can be used singly or together.

> **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*.

### 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 change involving the brightness attribute of *light.office1* and return the brightness attribute.

```
>>> 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 change involving *light.office1* changing from brightness 100 to 200 and return the brightness attribute.

```
>>> self.handle = self.listen_state(self.my_callback, "light.office_1", attribute
→= "brightness", 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)
```

**cancel_listen_state**(*self*, *handle*)

Cancels 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.

> **Parameters** `handle` – The handle returned when the `listen_state()` call was made.
>
> **Returns** None.

### Examples

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

**info_listen_state**(*self*, *handle*)

Gets information on state a callback from its handle.

> **Parameters** `handle` – The handle returned when the `listen_state()` call was made.
>
> **Returns** The values supplied for `entity`, `attribute`, and `kwargs` when the callback was initially created.

### Examples

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

## 8.2.2 Time

**parse_utc_string**(*self*, *utc_string*)

Converts a UTC to its string representation.

> **Parameters** `utc_string` (*str*) – A string that contains a date and time to convert.
>
> **Returns** An POSIX timestamp that is equivalent to the date and time contained in *utc_string*.

**get_tz_offset**(*self*)

Returns the timezone difference between UTC and Local Time in minutes.

**convert_utc**(*utc*)

Gets a *datetime* object for the specified 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. This function will accept one of these strings and convert it to a localised Python *datetime* object representing the timestamp.

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

**Returns** A localised Python *datetime* object representing the timestamp.

**sun_up**(*self*)

> Determines if the sun is currently up.

> **Returns** `True` if the sun is up, `False` otherwise.

> **Return type** bool

### Examples

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

**sun_down**(*self*)

> Determines if the sun is currently down.

> **Returns** `True` if the sun is down, `False` otherwise.

> **Return type** bool

### Examples

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

**parse_time**(*self*, *time_str*, *name=None*, *aware=False*)

> Creates a *time* object from its string representation.

> This functions takes a string representation of a time, or sunrise, or sunset offset and converts it to a datetime.time object.

> **Parameters**

>> • **time_str** (*str*) – A representation of the time in a string format with one of the following formats:

>>> a. `HH:MM:SS` - the time in Hours Minutes and Seconds, 24 hour format.

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

>> • **name** (*str, optional*) – Name of the calling app or module. It is used only for logging purposes.

>> • **aware** (*bool, optional*) – If `True` the created time object will be aware of timezone.

> **Returns** A *time* object, representing the time given in the *time_str* argument.

### Examples

```
>>> self.parse_time("17:30:00")
17:30:00
```

```
>>> time = self.parse_time("sunrise")
04:33:17
```

```
>>> time = self.parse_time("sunset + 00:30:00")
19:18:48
```

```
>>> time = self.parse_time("sunrise + 01:00:00")
05:33:17
```

**parse_datetime** (*self*, *time_str*, *name=None*, *aware=False*)

   Creates a *datetime* object from its string representation.

   This function takes a string representation of a date and time, or sunrise, or sunset offset and converts it to a *datetime* object.

   > **Parameters**
   >
   > - **time_str** (*str*) – A string representation of the datetime with one of the following formats:
   >
   >   a. `YY-MM-DD-HH:MM:SS` - the date and time in Year, Month, Day, Hours, Minutes, and Seconds, 24 hour format.
   >
   >   b. `HH:MM:SS` - the time in Hours Minutes and Seconds, 24 hour format.
   >
   >   c. `sunrise|sunset [+|- HH:MM:SS]` - time of the next sunrise or sunset with an optional positive or negative offset in Hours Minutes and seconds.
   >
   >   If the `HH:MM:SS` format is used, the resulting datetime object will have today's date.
   >
   > - **name** (*str, optional*) – Name of the calling app or module. It is used only for logging purposes.
   >
   > - **aware** (*bool, optional*) – If `True` the created datetime object will be aware of timezone.
   >
   > **Returns** A *datetime* object, representing the time and date given in the *time_str* argument.

   **Examples**

```
>>> self.parse_datetime("2018-08-09 17:30:00")
2018-08-09 17:30:00
```

```
>>> self.parse_datetime("17:30:00")
2019-08-15 17:30:00
```

```
>>> self.parse_datetime("sunrise")
2019-08-16 05:33:17
```

```
>>> self.parse_datetime("sunset + 00:30:00")
2019-08-16 19:18:48
```

```
>>> self.parse_datetime("sunrise + 01:00:00")
2019-08-16 06:33:17
```

**get_now** (*self*)

   Returns the current Local Date and Time.

---

**Examples**

```
>>> self.get_now()
2019-08-16 21:17:41.098813+00:00
```

**get_now_ts**(*self*)
    Returns the current Local Timestamp.

**Examples**

```
>>> self.get_now_ts()
1565990318.728324
```

**now_is_between**(*self*, *start_time*, *end_time*, *name=None*)
    Determines is the current *time* is within the specified start and end times.

    This function takes two string representations of a `time`, or `sunrise` or `sunset` offset and returns `true` if the current time is between those 2 times. Its implementation can correctly handle transitions across midnight.

    **Parameters**

    - **start_time** (`str`) – A string representation of the start time.

    - **end_time** (`str`) – A string representation of the end time.

    - **name** (`str, optional`) – Name of the calling app or module. It is used only for logging purposes.

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

    **Return type** bool

**Notes**

The string representation of the `start_time` and `end_time` should follows one of these formats:

    a. `HH:MM:SS` - the time in Hours Minutes and Seconds, 24 hour format.

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

**Examples**

```
>>> 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
```

**sunrise**(*self*, *aware=False*)
    Returns a *datetime* object that represents the next time Sunrise will occur.

    **Parameters aware** (`bool, optional`) – Specifies if the created datetime object will be *aware* of timezone or *not*.

### Examples

```
>>> self.sunrise()
2019-08-16 05:33:17
```

**sunset**(*self*, *aware=False*)

Returns a *datetime* object that represents the next time Sunset will occur.

> **Parameters aware** (`bool, optional`) – Specifies if the created datetime object will be *aware* of timezone or *not*.

### Examples

```
>>> self.sunset()
2019-08-16 19:48:48
```

**time**(*self*)

Returns a localised *time* object representing the current Local Time.

Use this in preference to the standard Python ways to discover the current time, especially when using the "Time Travel" feature for testing.

### Examples

```
>>> self.time()
20:15:31.295751
```

**datetime**(*self*, *aware=False*)

Returns a *datetime* object representing the current Local Date and Time.

Use this in preference to the standard Python ways to discover the current datetime, especially when using the "Time Travel" feature for testing.

> **Parameters aware** (`bool, optional`) – Specifies if the created datetime object will be *aware* of timezone or *not*.

### Examples

```
>>> self.datetime()
2019-08-15 20:15:55.549379
```

**date**(*self*)

Returns a localised *date* object representing the current Local Date.

Use this in preference to the standard Python ways to discover the current date, especially when using the "Time Travel" feature for testing.

### Examples

```
>>> self.date()
2019-08-15
```

**get_timezone**(*self*)
> Returns the current time zone.

## 8.2.3 Scheduler

**cancel_timer**(*self*, *handle*)
> Cancels a previously created timer.

> > **Parameters handle** – A handle value returned from the original call to create the timer.

> > **Returns** None.

> ### Examples

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

**info_timer**(*self*, *handle*)
> Gets information on a scheduler event from its handle.

> > **Parameters handle** – The handle returned when the scheduler call was made.

> > **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.

> > > or `None` - if handle is invalid or timer no longer exists.

> ### Examples

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

**run_in**(*self*, *callback*, *delay*, *\*\*kwargs*)
> Runs 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.

> > **Parameters**

> > > - **callback** – Function to be invoked when the requested state change occurs. It must conform to the standard Scheduler Callback format documented here.
> > > - **delay** (*int*) – Delay, in seconds before the callback is invoked.
> > > - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

> > **Keyword Arguments**

> > > - **random_start** (*int*) – Start of range of the random time.
> > > - **random_end** (*int*) – End of range of the random time.
> > > - **pin** (*bool, optional*) – If True, the callback will be pinned to a particular thread.
> > > - **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when
  it is invoked.

   **Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative
to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Run the specified callback after 10 seconds.

```
>>> self.handle = self.run_in(self.run_in_c, 10)
```

Run the specified callback after 10 seconds with a keyword arg (title).

```
>>> self.handle = self.run_in(self.run_in_c, 5, title = "run_in5")
```

**run_once** (*self*, *callback*, *start*, *\*\*kwargs*)
   Runs the callback once, at the specified time of day.

   **Parameters**

- **callback** – Function to be invoked at the specified time of day. It must conform to the
  standard Scheduler Callback format documented here.

- **start** – Should be either a Python `time` object or a `parse_time()` formatted string
  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.

- **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

   **Keyword Arguments**

- **random_start** (*int*) – Start of range of the random time.

- **random_end** (*int*) – End of range of the random time.

- **pin** (*bool, optional*) – If True, the callback will be pinned to a particular thread.

- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the call-
  back will be run by (0 - number of threads -1).

- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when
  it is invoked.

   **Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative
to denote a random offset before and event, or positive to denote a random offset after an event.

**Examples**

Run at 4pm today, or 4pm tomorrow if it is already after 4pm.

```
>>> runtime = datetime.time(16, 0, 0)
>>> handle = self.run_once(self.run_once_c, runtime)
```

Run today at 10:30 using the *parse_time()* function.

```
>>> handle = self.run_once(self.run_once_c, "10:30:00")
```

Run at sunset.

```
>>> handle = self.run_once(self.run_once_c, "sunset")
```

Run an hour after sunrise.

```
>>> handle = self.run_once(self.run_once_c, "sunrise + 01:00:00")
```

**run_at** (*self*, *callback*, *start*, *\*\*kwargs*)

Runs the callback once, at the specified time of day.

> **Parameters**
>
> - **callback** – Function to be invoked at the specified time of day. It must conform to the standard Scheduler Callback format documented here.
> - **start** – Should be either a Python `time` object or a `parse_time()` formatted string that specifies when the callback will occur.
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **random_start** (*int*) – Start of range of the random time.
> - **random_end** (*int*) – End of range of the random time.
> - **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.
> - **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).
> - **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.
>
> **Returns** A handle that can be used to cancel the timer.

**Notes**

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

The `run_at()` function will `raise` an exception if the specified time is in the `past`.

**Examples**

Run at 4pm today.

```
>>> runtime = datetime.time(16, 0, 0)
>>> today = datetime.date.today()
>>> event = datetime.datetime.combine(today, runtime)
>>> handle = self.run_at(self.run_at_c, event)
```

Run today at 10:30 using the *parse_time()* function.

```
>>> handle = self.run_at(self.run_at_c, "10:30:00")
```

Run on a specific date and time.

```
>>> handle = self.run_at(self.run_at_c, "2018-12-11 10:30:00")
```

Run at the next sunset.

```
>>> handle = self.run_at(self.run_at_c, "sunset")
```

Run an hour after the next sunrise.

```
>>> handle = self.run_at(self.run_at_c, "sunrise + 01:00:00")
```

**run_daily**(*self*, *callback*, *start*, *\*\*kwargs*)
    Runs the callback at the same time every day.

>    **Parameters**

>    - **callback** – Function to be invoked every day at the specified time. It must conform to the standard Scheduler Callback format documented here.

>    - **start** – Should be either a Python `time` object or a `parse_time()` formatted string 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. When specifying sunrise or sunset relative times using the `parse_datetime()` format, the time of the callback will be adjusted every day to track the actual value of sunrise or sunset.

>    - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

>    **Keyword Arguments**

>    - **random_start** (*int*) – Start of range of the random time.

>    - **random_end** (*int*) – End of range of the random time.

>    - **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.

>    - **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

>    - **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

>    **Returns**  A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Run daily at 7pm.

```
>>> runtime = datetime.time(19, 0, 0)
>>> self.run_daily(self.run_daily_c, runtime)
```

Run at 10:30 every day using the *parse_time()* function.

```
>>> handle = self.run_daily(self.run_daily_c, "10:30:00")
```

Run every day at sunrise.

```
>>> handle = self.run_daily(self.run_daily_c, "sunrise")
```

Run every day an hour after sunset.

```
>>> handle = self.run_daily(self.run_daily_c, "sunset + 01:00:00")
```

**run_hourly**(*self*, *callback*, *start*, *\*\*kwargs*)

Runs the callback at the same time every hour.

> #### Parameters
>
> - **callback** – Function to be invoked every hour at the specified time. It must conform to the standard Scheduler Callback format documented here.
> - **start** – 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.
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> #### Keyword Arguments
>
> - **random_start** (*int*) – Start of range of the random time.
> - **random_end** (*int*) – End of range of the random time.
> - **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.
> - **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).
> - **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.
>
> **Returns** A handle that can be used to cancel the timer.

#### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

#### Examples

Run every hour, on the hour.

```
>>> runtime = datetime.time(0, 0, 0)
>>> self.run_hourly(self.run_hourly_c, runtime)
```

**run_minutely**(*self*, *callback*, *start*, *\*\*kwargs*)

    Runs the callback at the same time every minute.

    **Parameters**

- **callback** – Function to be invoked every minute. It must conform to the standard Scheduler Callback format documented here.
- **start** – 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 minute` at the specified time. If time is not supplied, the callback will start a minute from the time that `run_minutely()` was executed.
- **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

    **Keyword Arguments**

- **random_start** (*int*) – Start of range of the random time.
- **random_end** (*int*) – End of range of the random time.
- **pin** (*bool, optional*) – If True, the callback will be pinned to a particular thread.
- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).
- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

    **Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Run every minute on the minute.

```
>>> time = datetime.time(0, 0, 0)
>>> self.run_minutely(self.run_minutely_c, time)
```

**run_every**(*self*, *callback*, *start*, *interval*, *\*\*kwargs*)

    Runs the callback with a configurable delay starting at a specific time.

    **Parameters**

- **callback** – Function to be invoked when the time interval is reached. It must conform to the standard Scheduler Callback format documented here.
- **start** – A Python `datetime` object that specifies when the initial callback will occur, or can take the *now* string alongside an added offset. If given in the past, it will be executed in the next interval time.
- **interval** – Frequency (expressed in seconds) in which the callback should be executed.

- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

**Keyword Arguments**

- **random_start** (*int*) – Start of range of the random time.
- **random_end** (*int*) – End of range of the random time.
- **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.
- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

**Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Run every 17 minutes starting in 2 hours time.

```
>>> self.run_every(self.run_every_c, time, 17 * 60)
```

Run every 10 minutes starting now.

```
>>> self.run_every(self.run_every_c, "now", 10 * 60)
```

Run every 5 minutes starting now plus 5 seconds.

```
>>> self.run_every(self.run_every_c, "now+5", 5 * 60)
```

**run_at_sunset** (*self*, *callback*, *\*\*kwargs*)
    Runs a callback every day at or around sunset.

**Parameters**

- **callback** – Function to be invoked at or around sunset. It must conform to the standard Scheduler Callback format documented here.
- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

**Keyword Arguments**

- **offset** (*int, optional*) – 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`.
- **random_start** (*int*) – Start of range of the random time.
- **random_end** (*int*) – End of range of the random time.
- **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.
- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

**Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Example using timedelta.

```
>>> self.run_at_sunset(self.sun, offset = datetime.timedelta(minutes = -45).total_
↪seconds())
```

Or you can just do the math yourself.

```
>>> self.run_at_sunset(self.sun, offset = 30 * 60)
```

Run at a random time +/- 60 minutes from sunset.

```
>>> self.run_at_sunset(self.sun, random_start = -60*60, random_end = 60*60)
```

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)
```

**run_at_sunrise**(*self*, *callback*, *\*\*kwargs*)
Runs a callback every day at or around sunrise.

#### Parameters

- **callback** – Function to be invoked at or around sunrise. It must conform to the standard Scheduler Callback format documented here.

- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

#### Keyword Arguments

- **offset** (*int, optional*) – 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`.

- **random_start** (*int*) – Start of range of the random time.

- **random_end** (*int*) – End of range of the random time.

- **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.

- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

**Returns** A handle that can be used to cancel the timer.

### Notes

The `random_start` value must always be numerically lower than `random_end` value, they can be negative to denote a random offset before and event, or positive to denote a random offset after an event.

### Examples

Run 45 minutes before sunset.

```
>>> self.run_at_sunrise(self.sun, offset = datetime.timedelta(minutes = -45).
↪total_seconds())
```

Or you can just do the math yourself.

```
>>> self.run_at_sunrise(self.sun, offset = 30 * 60)
```

Run at a random time +/- 60 minutes from sunrise.

```
>>> self.run_at_sunrise(self.sun, random_start = -60*60, random_end = 60*60)
```

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)
```

## 8.2.4 Service

**register_service**(*self*, *service*, *cb*, *\*\*kwargs*)

Registers a service that can be called from other apps, the REST API and the Event Stream

Using this function, an App can register a function to be available in the service registry. This will automatically make it available to other apps using the *call_service()* API call, as well as publish it as a service in the REST API and make it available to the *call_service* command in the event stream.

> **Parameters**
>
> - **service** – Name of the service, in the format *domain/service*. If the domain does not exist it will be created
> - **cb** – A reference to the function to be called when the service is requested. This function may be a regular function, or it may be async. Note that if it is an async function, it will run on AppDaemon's main loop meaning that any issues with the service could result in a delay of AppDaemon's core functions.
>
> **Returns** None

### Examples

```
>>> self.register_service("myservices/service1", mycallback)
```

**list_services**(*self*, *\*\*kwargs*)

List all services available within AD

Using this function, an App can request all available services within AD

> **Parameters** **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **\*\*kwargs** – Each service has different parameter requirements. This argument allows you to specify a comma-separated list of keyword value pairs, e.g., *namespace = global*.

- **namespace** (*str, optional*) – If a *namespace* is provided, AppDaemon will request the services within the given namespace. On the other hand, if no namespace is given, AppDaemon will use the last specified namespace or the default namespace. To get all services across AD, pass *global*. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.

> **Returns** All services within the requested namespace

### Examples

```
>>> self.list_services(namespace="global")
```

**call_service**(*self*, *service*, *\*\*kwargs*)

> Calls a Service within AppDaemon.

This function can call any service and provide any required parameters. By default, there are standard services that can be called within AD. Other services that can be called, are dependent on the plugin used, or those registered by individual apps using the *register_service* api. In a future release, all available services can be found using AD's Admin 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*.

> **Parameters**
>
> - **service** (*str*) – The service name.
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **\*\*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 all service calls require an `entity_id`.
> - **namespace** (*str, optional*) – If a *namespace* is provided, AppDaemon will change the state of the given entity in the given namespace. On the other hand, if no namespace is given, AppDaemon will use the last specified namespace or the default namespace. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.
>
> **Returns** Result of the *call_service* function if any

### Examples

HASS

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

MQTT

```
>>> call_service("mqtt/subscribe", topic="homeassistant/living_room/light", qos=2)
>>> call_service("mqtt/publish", topic="homeassistant/living_room/light", payload=
↪"on")
```

Utility

```
>>> call_service("app/restart", app="notify_app", namespace="appdaemon")
>>> call_service("app/stop", app="lights_app", namespace="appdaemon")
>>> call_service("app/reload", namespace="appdaemon")
```

For Utility, it is important that the *namespace* arg is set to `appdaemon` as no app can work within that *namespace*. If not namespace is specified, calling this function will rise an error.

## 8.2.5 Sequence

**run_sequence**(*self*, *sequence*, *\*\*kwargs*)
  Run an AppDaemon Sequence. Sequences are defined in a valid apps.yaml file or inline, and are sequences of service calls.

  **Parameters**

  - **sequence** – The sequence name, referring to the correct entry in apps.yaml, or a dict containing actual commands to run

  - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

  **Keyword Arguments namespace** (*str, optional*) – If a *namespace* is provided, AppDaemon will change the state of the given entity in the given namespace. On the other hand, if no namespace is given, AppDaemon will use the last specified namespace or the default namespace. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.

  **Returns** A handle that can be used with *cancel_sequence()* to terminate the script.

  **Examples**

  Run a yaml-defined sequence called "sequence.front_room_scene".

```
>>> handle = self.run_sequence("sequence.front_room_scene")
```

  Run an inline sequence.

```
>>> handle = self.run_sequence([{"light/turn_on": {"entity_id": "light.office_1"}}
↪, {"sleep": 5}, {"light.turn_off":
{"entity_id": "light.office_1"}}])
```

**cancel_sequence**(*self*, *handle*)
  Cancel an AppDaemon Sequence.

  **Parameters handle** – The handle returned by the *run_sequence()* call

  **Returns** None.

  **Examples**

```
>>> self.run_sequence(handle)
```

## 8.2.6 Events

**listen_event**(*self*, *callback*, *event=None*, *\*\*kwargs*)

Registers a callback for a specific event, or any event.

**Parameters**

- **callback** – Function to be invoked when the requested state change occurs. It must conform to the standard State Callback format documented here

- **event** (*optional*) – 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*) – Zero or more keyword arguments.

**Keyword Arguments**

- **oneshot** (*bool, optional*) – If `True`, the callback will be automatically cancelled after the first state change that results in a callback.

- **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter. The value `global` for namespace has special significance, and means that the callback will listen to state updates from any plugin.

- **pin** (*bool, optional*) – If `True`, the callback will be pinned to a particular thread.

- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

- **timeout** (*int, optional*) – If `timeout` is supplied as a parameter, the callback will be created as normal, but after `timeout` seconds, the callback will be removed.

- **\*\*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()* 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.

**Returns** A handle that can be used to cancel the callback.

### Examples

Listen all *"MODE_CHANGE"* events.

```
>>> 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)
```

**cancel_listen_event**(*self*, *handle*)
    Cancels a callback for a specific event.

> **Parameters handle** – A handle returned from a previous call to listen_event().
>
> **Returns** None.

**Examples**

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

**info_listen_event**(*self*, *handle*)
    Gets information on an event callback from its handle.

> **Parameters handle** – The handle returned when the listen_event() call was made.
>
> **Returns** The values (service, kwargs) supplied when the callback was initially created.

**Examples**

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

**fire_event**(*self*, *event*, *\*\*kwargs*)
    Fires an event on the AppDaemon bus, for apps and plugins.

> **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** (`optional`) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **namespace** (`str, optional`) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases, it is safe to ignore this parameter.
> - **\*\*kwargs** (`optional`) – Zero or more keyword arguments that will be supplied as part of the event.
>
> **Returns** None.

**Examples**

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

## 8.2.7 Logging

**log**(*self*, *msg*, *\*args*, *\*\*kwargs*)
    Logs a message to AppDaemon's main logfile.

> **Parameters**

- **msg** (`str`) – The message to log.

- **\*\*kwargs** (`optional`) – Zero or more keyword arguments.

**Keyword Arguments**

- **level** (`str, optional`) – The log level of the message - takes a string representing the standard logger levels (Default: `"WARNING"`).

- **ascii_encode** (`bool, optional`) – Switch to disable the encoding of all log messages to ascii. Set this to true if you want to log UTF-8 characters (Default: `True`).

- **log** (`str, optional`) – Send the message to a specific log, either system or user_defined. System logs are `main_log`, `error_log`, `diag_log` or `access_log`. Any other value in use here must have a corresponding user-defined entity in the `logs` section of appdaemon.yaml.

- **stack_info** (`bool, optional`) – If `True` the stack info will included.

**Returns** None.

### Examples

Log a message to the main logfile of the system.

```
>>> self.log("Log Test: Parameter is %s", some_variable)
```

Log a message to the specified logfile.

```
>>> self.log("Log Test: Parameter is %s", some_variable, log="test_log")
```

Log a message with error-level to the main logfile of the system.

```
>>> self.log("Log Test: Parameter is %s", some_variable, level = "ERROR")
```

Log a message using *placeholders* to the main logfile of the system.

```
>>> self.log("Line: __line__, module: __module__, function: __function__, Msg:
→Something bad happened")
```

Log a WARNING message (including the stack info) to the main logfile of the system.

```
>>> self.log("Stack is", some_value, level="WARNING", stack_info=True)
```

**error** (*self*, *msg*, *\*args*, *\*\*kwargs*)
Logs a message to AppDaemon's error logfile.

**Parameters**

- **msg** (`str`) – The message to log.

- **\*\*kwargs** (`optional`) – Zero or more keyword arguments.

**Keyword Arguments**

- **level** (`str, optional`) – The log level of the message - takes a string representing the standard logger levels.

- **ascii_encode** (`bool, optional`) – Switch to disable the encoding of all log messages to ascii. Set this to true if you want to log UTF-8 characters (Default: `True`).

- **log** (*str, optional*) – Send the message to a specific log, either system or user_defined. System logs are main_log, error_log, diag_log or access_log. Any other value in use here must have a corresponding user-defined entity in the logs section of appdaemon.yaml.

**Returns** None.

### Examples

Log an error message to the error logfile of the system.

```
>>> self.error("Some Warning string")
```

Log an error message with critical-level to the error logfile of the system.

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

**listen_log**(*self, callback, level='INFO', **kwargs*)

Registers the App to receive a callback every time an App logs a message.

#### Parameters

- **callback** (*function*) – Function to be called when a message is logged.
- **level** (*str*) – Logging level to be used - lower levels will not be forwarded to the app (Default: "INFO").
- ***kwargs** (*optional*) – Zero or more keyword arguments.

#### Keyword Arguments

- **log** (*str, optional*) – Name of the log to listen to, default is all logs. The name should be one of the 4 built in types main_log, error_log, diag_log or access_log or a user defined log entry.
- **pin** (*bool, optional*) – If True, the callback will be pinned to a particular thread.
- **pin_thread** (*int, optional*) – Specify which thread from the worker pool the callback will be run by (0 - number of threads -1).

**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.

### Examples

Listen to all WARNING log messages of the system.

```
>>> self.handle = self.listen_log(self.cb, "WARNING")
```

Listen to all WARNING log messages of the *main_log*.

```
>>> self.handle = self.listen_log(self.cb, "WARNING", log="main_log")
```

Listen to all WARNING log messages of a user-defined logfile.

```
>>> self.handle = self.listen_log(self.cb, "WARNING", log="my_custom_log")
```

**cancel_listen_log**(*self*, *handle*)
: Cancels the log callback for the App.

> **Parameters** **handle** – The handle returned when the *listen_log* call was made.
>
> **Returns** None.

### Examples

```
>>> self.cancel_listen_log(handle)
```

**get_main_log**(*self*)
: Returns the underlying logger object used for the main log.

### Examples

Log a critical message to the *main* logfile of the system.

```
>>> log = self.get_main_log()
>>> log.critical("Log a critical error")
```

**get_error_log**(*self*)
: Returns the underlying logger object used for the error log.

### Examples

Log an error message to the *error* logfile of the system.

```
>>> error_log = self.get_error_log()
>>> error_log.error("Log an error", stack_info=True, exc_info=True)
```

**get_user_log**(*self*, *log*)
: Gets the specified-user logger of the App.

> **Parameters** **log** (*str*) – The name of the log you want to get the underlying logger object from,
> as described in the logs section of appdaemon.yaml.
>
> **Returns** The underlying logger object used for the error log.

### Examples

Log an error message to a user-defined logfile.

```
>>> log = self.get_user_log("test_log")
>>> log.error("Log an error", stack_info=True, exc_info=True)
```

**set_log_level**(*self*, *level*)
: Sets a specific log level for the App.

> **Parameters** **level** (*str*) – Log level.
>
> **Returns** None.

### Notes

Supported log levels: `INFO`, `WARNING`, `ERROR`, `CRITICAL`, `DEBUG`, `NOTSET`.

### Examples

```
>>> self.set_log_level("DEBUG")
```

**set_error_level**(*self*, *level*)
  Sets the log level to send to the *error* logfile of the system.

> **Parameters** **level** (*str*) – Error level.
>
> **Returns** None.

### Notes

Supported log levels: `INFO`, `WARNING`, `ERROR`, `CRITICAL`, `DEBUG`, `NOTSET`.

## 8.2.8 Dashboard

**dash_navigate**(*self*, *target*, *timeout=-1*, *ret=None*, *sticky=0*)
  Forces all connected Dashboards to navigate to a new URL.

> **Parameters**
>
> - **target** (*str*) – Name of the new Dashboard to navigate to (e.g., `/SensorPanel`). Note that this value is not a URL.
>
> - **timeout** (*int*) – 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.
>
> - **ret** (*str*) – Dashboard to return to after the timeout has elapsed.
>
> - **sticky** (*int*) – Specifies whether or not to return to the original dashboard after it has been clicked on. The default behavior (`sticky=0`) is to remain on the new dashboard if clicked, or return to the original otherwise. By using a different value (sticky= 5), clicking the dashboard will extend the amount of time (in seconds), but it will return to the original dashboard after a period of inactivity equal to timeout.
>
> **Returns** None.

### Examples

Switch to AlarmStatus Panel then return to current panel after 10 seconds.

```
>>> self.dash_navigate("/AlarmStatus", timeout=10)
```

Switch to Locks Panel then return to Main panel after 10 seconds.

```
>>> self.dash_navigate("/Locks", timeout=10, ret="/SensorPanel")
```

## 8.2.9 Namespace

**set_namespace** (*self*, *namespace*)
> Sets a new namespace for the App to use from that point forward.

> > **Parameters namespace** (`str`) – Name of the new namespace

> > **Returns** None.

### Examples

```
>>> self.set_namespace("hass1")
```

**get_namespace** (*self*)
> Returns the App's namespace.

**list_namespaces** (*self*)
> Returns a list of available namespaces.

### Examples

```
>>> self.list_namespaces()
```

**save_namespace** (*self*, *\*\*kwargs*)
> Saves entities created in user-defined namespaces into a file.

> This way, when AD restarts these entities will be reloaded into AD with its previous states within the namespace. This can be used as a basic form of non-volatile storage of entity data. Depending on the configuration of the namespace, this function can be setup to constantly be running automatically or only when AD shutdown. This function also allows for users to manually execute the command as when needed.

> > **Parameters \*\*kwargs** (`optional`) – Zero or more keyword arguments.

> > **Keyword Arguments namespace** (`str, optional`) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

> > **Returns** None.

### Examples

Save all entities of the default namespace.

```
>>> self.save_namespace()
```

## 8.2.10 Services

Note: A service call always uses the app's default namespace. Although namespaces allow a new and easy way to work with multiple namespaces from within a single App, it is essential to understand how they work before using them in service's calls. See the section on namespaces for a detailed description.

AppDaemon has a predefined list of namespaces that can be used only for particular services. Listed below are the services by namespace.

`appdaemon` namespace only:

**app/start**

Starts an app that has been terminated. The *app* name arg is required.

```
>>> self.call_service("app/start", app="light_app", namespace="appdaemon")
```

**app/stop**

Stops a running app. The *app* name arg is required.

```
>>> self.call_service("app/stop", app="light_app", namespace="appdaemon")
```

**app/restart**

Restarts a running app. This service basically stops and starts the app. The *app* name arg is required.

```
>>> self.call_service("app/restart", app="light_app", namespace="appdaemon")
```

**app/reload**

Checks for an app update. Useful if AD is running in production mode, and app changes need to be checked and loaded.

```
>>> self.call_service("app/reload", namespace="appdaemon")
```

**production_mode/set**

Sets the production mode AD is running on. The value of the *mode* arg has to be *True* or *False*.

```
>>> self.call_service("production_mode/set", mode=True, namespace="appdaemon")
```

All namespaces except `appdaemon`, `global`, and `admin`:

**state/add_entity**

Adds an existing entity to the required namespace.

```
>>> self.call_service("state/set", entity_id="sensor.test", state="on", attributes={
→"friendly_name" : "Sensor Test"}, namespace="default")
```

**state/set**

Sets the state of an entity. This service allows any key-worded args to define what entity's values need to be set.

```
>>> self.call_service("state/set", entity_id="sensor.test", state="on", attributes={
→"friendly_name" : "Sensor Test"}, namespace="default")
```

**state/remove_entity**

Removes an existing entity from the required namespace.

```
>>> self.call_service("state/remove_entity", entity_id="sensor.test"}, namespace=
→"default")
```

All namespaces except `appdaemon`:

**event/fire**

Fires an event within the specified namespace. The *event* arg is required.

```
>>> self.call_service("event/fire", event="test_event", entity_id="appdaemon.test",
→namespace="hass")
```

`rules` namespace only:

**sequence/run**

Runs a predefined sequence. The *entity_id* arg with the sequence full-qualified entity name is required.

```
>>> self.call_service("sequence/run", entity_id ="sequence.christmas_lights",
↪namespace="rules")
```

## 8.2.11 Threading

**set_app_pin**(*self*, *pin*)

Sets an App to be pinned or unpinned.

> **Parameters** **pin** (*bool*) – Sets whether the App becomes pinned or not.
>
> **Returns** None.

### Examples

The following line should be put inside the *initialize()* function.

```
>>> self.set_app_pin(True)
```

**get_app_pin**(*self*)

Finds out if the current App is currently pinned or not.

> **Returns** True if the App is pinned, False otherwise.
>
> **Return type** bool

### Examples

```
>>> if self.get_app_pin(True):
>>>     self.log("App pinned!")
```

**set_pin_thread**(*self*, *thread*)

Sets the thread that the App will be pinned to.

> **Parameters** **thread** (*int*) – Number of the thread to pin to. Threads start at 0 and go up to the number of threads specified in `appdaemon.yaml` -1.
>
> **Returns** None.

### Examples

The following line should be put inside the *initialize()* function.

```
>>> self.set_pin_thread(5)
```

**get_pin_thread**(*self*)

Finds out which thread the App is pinned to.

> **Returns** The thread number or -1 if the App is not pinned.
>
> **Return type** int

---

**Examples**

```
>>> thread = self.get_pin_thread():
>>> self.log(f"I'm pinned to thread: {thread}")
```

## 8.2.12 Async

**create_task** (*self*, *coro*, *callback=None*, *\*\*kwargs*)

Schedules a Coroutine to be executed.

> **Parameters**
>
> > • **coro** – The coroutine object (*not coroutine function*) to be executed.
> >
> > • **callback** – The non-async callback to be executed when complete.
> >
> > • **\*\*kwargs** (*optional*) – Any additional keyword arguments to send the callback.
>
> **Returns** A Future, which can be cancelled by calling f.cancel().

**Examples**

```
>>> f = self.create_task(asyncio.sleep(3), callback=self.coro_callback)
>>>
>>> def coro_callback(self, kwargs):
```

**run_in_executor** (*self*, *func*, *\*args*, *\*\*kwargs*)

**sleep** (*delay*, *result=None*)

Pause execution for a certain time span (not available in sync apps)

> **Parameters**
>
> > • **delay** (*int*) – Number of seconds to pause.
> >
> > • **result** (*optional*) – Result to return upon delay completion.
>
> **Returns** Result or *None*.

**Notes**

This function is not available in sync apps.

**Examples**

```
>>> async def myfunction(self):
>>>     await self.sleep(5)
```

## 8.2.13 Utility

**get_app** (*self*, *name*)

Gets 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.

> **Parameters** **name** (*str*) – 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.

### Examples

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

**get_ad_version** ()
: Returns a string with the current version of AppDaemon.

### Examples

```
>>> version = self.get_ad_version()
```

**entity_exists** (*self*, *entity_id*, *\*\*kwargs*)
: Checks the existence of an entity in Home Assistant.

When working with multiple Home Assistant instances, it is possible to specify the namespace, so that it checks within the right instance in in the event the app is working in a different instance. Also when using this function, it is also possible to check if an AppDaemon entity exists.

> **Parameters**
>
> - **entity_id** (*str*) – The fully qualified entity id (including the device type).
>
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments** **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns** `True` if the entity id exists, `False` otherwise.
>
> **Return type** bool

### Examples

Check if the entity light.living_room exist within the app's namespace

```
>>> if self.entity_exists("light.living_room"):
>>>     #do something
```

Check if the entity mqtt.security_settings exist within the *mqtt* namespace if the app is operating in a different namespace like default

```
>>> if self.entity_exists("mqtt.security_settings", namespace = "mqtt"):
>>>     #do something
```

**split_entity** (*self*, *entity_id*, *\*\*kwargs*)
: Splits an entity into parts.

This utility function 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.

> **Parameters**

- **entity_id** (*str*) – The fully qualified entity id (including the device type).
- **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

**Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

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

### Examples

Do some action if the device of the entity is *scene*.

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

**remove_entity** (*self*, *entity_id*, *\*\*kwargs*)

Deletes an entity created within a namespaces.

If an entity was created, and its deemed no longer needed, by using this function, the entity can be removed from AppDaemon permanently.

#### Parameters

- **entity_id** (*str*) – The fully qualified entity id (including the device type).
- **\*\*kwargs** (*optional*) – Zero or more keyword arguments.

**Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

**Returns** None.

### Examples

Delete the entity in the present namespace.

```
>>> self.remove_entity('sensor.living_room')
```

Delete the entity in the *mqtt* namespace.

```
>>> self.remove_entity('mqtt.living_room_temperature', namespace = 'mqtt')
```

**split_device_list** (*devices*)

Converts a comma-separated list of device types to 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.

**Parameters devices** (*str*) – A comma-separated list of devices to be split (without spaces).

**Returns** A list of split devices with 1 or more entries.

**Examples**

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

**get_plugin_config**(*self*, *\*\*kwargs*)

Gets any useful metadata that the plugin may have available.

For instance, for the HASS plugin, this will return Home Assistant configuration data such as latitude and longitude.

> **Parameters** **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments** **namespace** (*str*) – Select the namespace of the plugin for which data is desired.
>
> **Returns** A dictionary containing all the configuration information available from the Home Assistant `/api/config` endpoint.

**Examples**

```
>>> config = self.get_plugin_config()
>>> self.log(f'My current position is {config["latitude"]}(Lat), {config[
↪"longitude"]}(Long)')
My current position is 50.8333(Lat), 4.3333(Long)
```

**friendly_name**(*self*, *entity_id*, *\*\*kwargs*)

Gets the Friendly Name of an entity.

> **Parameters**
>
> - **entity_id** (*str*) – The fully qualified entity id (including the device type).
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments** **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns** The friendly name of the entity if it exists or the entity id if not.
>
> **Return type** str

**Examples**

```
>>> tracker = "device_tracker.andrew"
>>> friendly_name = self.friendly_name(tracker)
>>> tracker_state = self.get_tracker_state(tracker)
>>> self.log(f"{tracker}  ({friendly_name}) is {tracker_state}.")
device_tracker.andrew (Andrew Tracker) is on.
```

**set_production_mode**(*self*, *mode=True*)

Deactivates or activates the production mode in AppDaemon.

When called without declaring passing any arguments, mode defaults to `True`.

> **Parameters** **mode** (*bool*) – If it is `True` the production mode is activated, or deactivated otherwise.
>
> **Returns** The specified mode or `None` if a wrong parameter is passed.

**start_app**(*self*, *app*, *\*\*kwargs*)

> Starts an App which can either be running or not.
>
> This Api call cannot start an app which has already been disabled in the App Config. It essentially only runs the initialize() function in the app, and changes to attributes like class name or app config is not taken into account.
>
> > **Parameters**
> >
> > - **app** (*str*) – Name of the app.
> >
> > - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
> >
> > **Returns** None.

### Examples

```
>>> self.start_app("lights_app")
```

**stop_app**(*self*, *app*, *\*\*kwargs*)

> Stops an App which is running.
>
> > **Parameters**
> >
> > - **app** (*str*) – Name of the app.
> >
> > - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
> >
> > **Returns** None.

### Examples

```
>>> self.stop_app("lights_app")
```

**restart_app**(*self*, *app*, *\*\*kwargs*)

> Restarts an App which can either be running or not.
>
> > **Parameters**
> >
> > - **app** (*str*) – Name of the app.
> >
> > - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
> >
> > **Returns** None.

### Examples

```
>>> self.restart_app("lights_app")
```

**reload_apps**(*self*, *\*\*kwargs*)

> Reloads the apps, and loads up those that have changes made to their .yaml or .py files.
>
> This utility function can be used if AppDaemon is running in production mode, and it is needed to reload apps that changes have been made to.
>
> > **Parameters** **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
> >
> > **Returns** None.

**Examples**

```
>>> self.reload_apps()
```

## 8.2.14 Dialogflow

**get_dialogflow_intent** (*self*, *data*)
    Gets the intent's action from the Google Home response.

> **Parameters data** – Response received from Google Home.
>
> **Returns** A string representing the Intent from the interaction model that was requested, or `None`, if no action was received.

**Examples**

```
>>> intent = ADAPI.get_dialogflow_intent(data)
```

**get_dialogflow_slot_value** (*data*, *slot=None*)
    Gets slots' values from the interaction model.

> **Parameters**
>
> • **data** – Response received from Google Home.
>
> • **slot** (*str*) – Name of the slot. If a name is not specified, all slots will be returned as a dictionary. If a name is specified but is not found, `None` will be returned.
>
> **Returns** A string representing the value of the slot from the interaction model, or a hash of slots.

**Examples**

```
>>> beer_type = ADAPI.get_dialogflow_intent(data, "beer_type")
>>> all_slots = ADAPI.get_dialogflow_intent(data)
```

**format_dialogflow_response** (*self*, *speech=None*)
    Formats a response to be returned to Google Home, including speech.

> **Parameters speech** (*str*) – The text for Google Home to say.
>
> **Returns** None.

**Examples**

```
>>> ADAPI.format_dialogflow_response(speech = "Hello World")
```

## 8.2.15 Alexa

**get_alexa_intent** (*data*)
    Gets the Intent's name from the Alexa response.

> **Parameters data** – Response received from Alexa.

**Returns** A string representing the Intent's name from the interaction model that was requested, or `None`, if no Intent was received.

### Examples

```
>>> intent = ADAPI.get_alexa_intent(data)
```

**get_alexa_slot_value**(*data*, *slot=None*)

Gets values for slots from the interaction model.

> **Parameters**
>
> - **data** – The request data received from Alexa.
>
> - **slot** – Name of the slot. If a name is not specified, all slots will be returned as a dictionary. If a name is specified but is not found, None will be returned.
>
> **Returns** A `string` representing the value of the slot from the interaction model, or a `hash` of slots.

### Examples

```
>>> beer_type = ADAPI.get_alexa_intent(data, "beer_type")
>>> all_slots = ADAPI.get_alexa_intent(data)
```

**format_alexa_response**(*speech=None*, *card=None*, *title=None*)

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

> **Parameters**
>
> - **speech** (*str*) – The text for Alexa to say.
>
> - **card** (*str*) – Text for the card.
>
> - **title** (*str*) – Title for the card.
>
> **Returns** None.

### Examples

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

**get_alexa_error**(*data*)

Gets the error message from the Alexa API response.

> **Parameters data** – Response received from the Alexa API .
>
> **Returns** A string representing the value of message, or `None` if no error message was received.

## 8.2.16 API

**register_endpoint**(*self*, *callback*, *name=None*)

Registers an endpoint for API calls into the current App.

> **Parameters**

- **callback** – The function to be called when a request is made to the named endpoint.
- **name** (*str, optional*) – The name of the endpoint to be used for the call (Default: `None`).

**Returns** A handle that can be used to remove the registration.

### Examples

It should be noted that the register function, should return a string (can be empty), and an HTTP OK status response (e.g., *200*. If this is not added as a returned response, the function will generate an error each time it is processed.

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

**unregister_endpoint** (*self*, *handle*)

Removes a previously registered endpoint.

**Parameters handle** – A handle returned by a previous call to `register_endpoint`

**Returns** None.

### Examples

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

WebRoute ~~~

**register_route** (*self*, *callback*, *route=None*, *\*\*kwargs*)

**Registers a route for Web requests into the current App.** By registering an app web route, this allows to make use of AD's internal web server to serve web clients. All routes registered using this api call, can be accessed using `http://AD_IP:Port/app/route`.

**Parameters**

- **callback** – The function to be called when a request is made to the named route. This must be an async function
- **route** (*str, optional*) – The name of the route to be used for the request (Default: the app's name).

**Keyword Arguments**

- **token** (*str, optional*) – A previously registered token can be passed with the api call, which
- **be used to secure the app route. This allows for different security credentials to be used across different** (*can*) –
- **routes. It should be noted that if a device has already registered using AD's Admin UI's password** (*app*) –
- **a cookie has been stored by the browser, that device will bypass the token and still access the web server.** (*and*) –

**Returns** A handle that can be used to remove the registration.

---

**Examples**

It should be noted that the register function, should return a string (can be empty), and an HTTP OK status response (e.g., *200*. If this is not added as a returned response, the function will generate an error each time it is processed.

```
>>> self.register_route(my_callback)
>>> self.register_route(stream_cb, "camera")
```

**unregister_route**(*self*, *handle*)
    Removes a previously registered app route.

        **Parameters** `handle` – A handle returned by a previous call to `register_app_route`

        **Returns** None.

**Examples**

```
>>> self.unregister_route(handle)
```

## 8.2.17 Other

**run_in_thread**(*self*, *callback*, *thread*, *\*\*kwargs*)
    Schedules a callback to be run in a different thread from the current one.

        **Parameters**

- **callback** – Function to be run on the new thread.
- **thread**(*int*) – Thread number (0 - number of threads).
- **\*\*kwargs** – Arbitrary keyword parameters to be provided to the callback function when it is invoked.

        **Returns** None.

**Examples**

```
>>> self.run_in_thread(my_callback, 8)
```

**get_thread_info**(*self*)
    Gets information on AppDaemon worker threads.

        **Returns** A dictionary containing all the information for AppDaemon worker threads.

**Examples**

```
>>> thread_info = self.get_thread_info()
```

**get_scheduler_entries**(*self*)
    Gets information on AppDaemon scheduler entries.

        **Returns** A dictionary containing all the information for entries in the AppDaemon scheduler.

**Examples**

```
>>> schedule = self.get_scheduler_entries()
```

**get_callback_entries**(*self*)

Gets information on AppDaemon callback entries.

> **Returns** A dictionary containing all the information for entries in the AppDaemon state, and event callback table.

**Examples**

```
>>> callbacks = self.get_callback_entries()
```

**depends_on_module**(*self*, *\*modules*)

Registers a global_modules dependency for an app.

> **Parameters** **\*modules** – Modules to register a dependency on.

> **Returns** None.

**Examples**

```
>>> import somemodule
>>> import anothermodule
>>> # later
>>> self.depends_on_module([somemodule)
```

# HASS API Reference

This page provides a list of API calls and specific information related to the HASS plugin.

## 9.1 App Creation

To create apps based on just the AppDaemon base API, use some code like the following:

```python
import hassapi as hass

class MyApp(hass.Hass):

    def initialize(self):
```

## 9.2 Reference

## 9.3 Services

**turn_on**(*self*, *entity_id*, *\*\*kwargs*)
  Turns *on* a Home Assistant entity.

  This is a convenience function for the `homeassistant.turn_on` function. It can turn `on` pretty much anything in Home Assistant that can be turned `on` or `run` (e.g., *Lights*, *Switches*, *Scenes*, *Scripts*, etc.).

  **Parameters**

  - **entity_id** (*str*) – Fully qualified id of the thing to be turned `on` (e.g., *light.office_lamp*, *scene.downstairs_on*).

  - **\*\*kwargs** – Zero or more keyword arguments.

  **Returns** None.

### Examples

Turn *on* a switch.

```
>>> self.turn_on("switch.backyard_lights")
```

Turn *on* a scene.

```
>>> self.turn_on("scene.bedroom_on")
```

Turn *on* a light and set its color to green.

```
>>> self.turn_on("light.office_1", color_name = "green")
```

**turn_off**(*self*, *entity_id*, *\*\*kwargs*)
: Turns *off* a Home Assistant entity.

This is a convenience function for the `homeassistant.turn_off` function. It can turn `off` pretty much anything in Home Assistant that can be turned `off` (e.g., *Lights*, *Switches*, etc.).

> **Parameters**
> - **entity_id** (*str*) – Fully qualified id of the thing to be turned `off` (e.g., *light.office_lamp*, *scene.downstairs_on*).
> - **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

### Examples

Turn *off* a switch.

```
>>> self.turn_off("switch.backyard_lights")
```

Turn *off* a scene.

```
>>> self.turn_off("scene.bedroom_on")
```

**toggle**(*self*, *entity_id*, *\*\*kwargs*)
: Toggles between `on` and `off` for the selected entity.

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

> **Parameters**
> - **entity_id** (*str*) – Fully qualified id of the thing to be turned `off` (e.g., *light.office_lamp*, *scene.downstairs_on*).
> - **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

### Examples

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

**set_value**(*self*, *entity_id*, *value*, *\*\*kwargs*)

Sets the value of an *input_number*.

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

> **Parameters**
>
> - **entity_id** (`str`) – Fully qualified id of *input_number* to be changed (e.g., *input_number.alarm_hour*).
>
> - **value** (`int or float`) – The new value to set the *input_number* to.
>
> - **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

### Examples

```
>>> self.set_value("input_number.alarm_hour", 6)
```

**set_textvalue**(*self*, *entity_id*, *value*, *\*\*kwargs*)

Sets the value of an *input_text*.

This is a convenience function for the `input_text.set_value` function. It can set the value of an *input_text* in Home Assistant.

> **Parameters**
>
> - **entity_id** (`str`) – Fully qualified id of *input_text* to be changed (e.g., *input_text.text1*).
>
> - **value** (`str`) – The new value to set the *input_text* to.
>
> - **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

### Examples

```
>>> self.set_textvalue("input_text.text1", "hello world")
```

**select_option**(*self*, *entity_id*, *option*, *\*\*kwargs*)

Sets the value of an *input_option*.

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

> **Parameters**
>
> - **entity_id** (`str`) – Fully qualified id of *input_select* to be changed (e.g., *input_select.mode*).
>
> - **option** (`str`) – The new value to set the *input_select* to.
>
> - **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

**Examples**

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

**notify** (*self*, *message*, *\*\*kwargs*)

Sends a notification.

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`.

> **Parameters**
>> • **message** (*str*) – Message to be sent to the notification service.
>>
>> • **\*\*kwargs** – Zero or more keyword arguments.
>
> **Returns** None.

**Examples**

```
>>> self.notify("Switching mode to Evening")
>>> self.notify("Switching mode to Evening", title = "Some Subject", name = "smtp
↪")
    # will send a message through notify.smtp instead of the default notify.notify
```

**render_template** (*self*, *template*, *\*\*kwargs*)

Renders a Home Assistant Template

> **Parameters template** (*str*) – The Home Assistant Template to be rendered.
>
> **Keyword Arguments None.** –
>
> **Returns** The rendered template in a native Python type.

**Examples**

```
>>> self.render_template("{{ states('sun.sun') }}")
Returns (str) above_horizon
```

```
>>> self.render_template("{{ is_state('sun.sun', 'above_horizon') }}")
Returns (bool) True
```

```
>>> self.render_template("{{ states('sensor.outside_temp') }}")
Returns (float) 97.2
```

# 9.4 Presence

**get_trackers** (*self*, *\*\*kwargs*)

Returns a list of all device tracker names.

> **Parameters \*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments**

- **person** (*boolean, optional*) – If set to True, use person rather than device_tracker as the device type to query

- **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

### Examples

```
>>> trackers = self.get_trackers()
>>> for tracker in trackers:
>>>     do something
>>> people = self.get_trackers(person=True)
>>> for person in people:
>>>     do something
```

**get_tracker_details**(*self*, *\*\*kwargs*)

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

> **Parameters \*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **person** (*boolean, optional*) – If set to True, use person rather than device_tracker as the device type to query
>
> - **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

### Examples

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

**get_tracker_state**(*self*, *entity_id*, *\*\*kwargs*)

Gets the state of a tracker.

> **Parameters entity_id** – Fully qualified entity id of the device tracker or person to query, e.g., device_tracker.andrew or person.andrew.
>
> **Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns**
>
> 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.

**Examples**

```
>>> state = self.get_tracker_state("device_tracker.andrew")
>>>     self.log("state is {}".format(state))
>>> state = self.get_tracker_state("person.andrew")
>>>     self.log("state is {}".format(state))
```

**anyone_home**(*self*, *\*\*kwargs*)

    Determines if the house/apartment is occupied.

    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.

        **Parameters** `**kwargs` (*optional*) – Zero or more keyword arguments.

        **Keyword Arguments**

            • **person** (*boolean, optional*) – If set to True, use person rather than device_tracker as the device type to query

            • **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

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

**Examples**

```
>>> if self.anyone_home():
>>>     do something
>>> if self.anyone_home(person=True):
>>>     do something
```

**everyone_home**(*self*, *\*\*kwargs*)

    Determine if all family's members at 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.

        **Parameters** `**kwargs` (*optional*) – Zero or more keyword arguments.

        **Keyword Arguments**

            • **person** (*boolean, optional*) – If set to True, use person rather than device_tracker as the device type to query

            • **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

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

**Examples**

```
>>> if self.everyone_home():
>>>     do something
>>> if self.everyone_home(person=True):
>>>     do something
```

**noone_home**(*self*, *\*\*kwargs*)

> Determines if the house/apartment is empty.
>
> 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.
>
> > **Parameters** **\*\*kwargs** (`optional`) – Zero or more keyword arguments.
> >
> > **Keyword Arguments**
> >
> > - **person** (`boolean, optional`) – If set to True, use person rather than device_tracker as the device type to query
> >
> > - **namespace** (`str, optional`) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.
> >
> > **Returns** Returns `True` if no one is home, `False` otherwise.

**Examples**

```
>>> if self.noone_home():
>>>     do something
>>> if self.noone_home(person=True):
>>>     do something
```

## 9.5 Database

**get_history**(*self*, *\*\*kwargs*)

> Gets access to the HA Database. This is a convenience function that allows accessing the HA Database, so the history state of a device can be retrieved. It allows for a level of flexibility when retrieving the data, and returns it as a dictionary list. Caution must be taken when using this, as depending on the size of the database, it can take a long time to process.
>
> > **Parameters** **\*\*kwargs** (`optional`) – Zero or more keyword arguments.
> >
> > **Keyword Arguments**
> >
> > - **entity_id** (`str, optional`) – Fully qualified id of the device to be querying, e.g., `light.office_lamp` or `scene.downstairs_on` This can be any entity_id in the database. If this is left empty, the state of all entities will be retrieved within the specified time. If both `end_time` and `start_time` explained below are declared, and `entity_id` is specified, the specified `entity_id` will be ignored and the history states of *all* entity_id in the database will be retrieved within the specified time.
> >
> > - **days** (`int, optional`) – The days from the present-day walking backwards that is required from the database.
> >
> > - **start_time** (`optional`) – The start time from when the data should be retrieved. This should be the furthest time backwards, like if we wanted to get data from now until two days ago. Your start time will be the last two days datetime. `start_time` time can be either a UTC aware time string like `2019-04-16 12:00:03+01:00` or a `datetime.datetime` object.
> >
> > - **end_time** (`optional`) – The end time from when the data should be retrieved. This should be the latest time like if we wanted to get data from now until two days ago. Your end time will be today's datetime `end_time` time can be either a UTC aware time string like `2019-04-16 12:00:03+01:00` or a `datetime.datetime` object. It should

be noted that it is not possible to declare only `end_time`. If only `end_time` is declared without `start_time` or `days`, it will revert to default to the latest history state. When `end_time` is specified, it is not possible to declare `entity_id`. If `entity_id` is specified, `end_time` will be ignored.

- **callback** (*callable, optional*) – If wanting to access the database to get a large amount of data, using a direct call to this function will take a long time to run and lead to AD cancelling the task. To get around this, it is better to pass a function, which will be responsible of receiving the result from the database. The signature of this function follows that of a scheduler call.

- **namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

**Returns** An iterable list of entity_ids and their history state.

### Examples

Get device state over the last 5 days.

```
>>> data = self.get_history("light.office_lamp", days = 5)
```

Get device state over the last 2 days and walk forward.

```
>>> import datetime
>>> from datetime import timedelta
>>> start_time = datetime.datetime.now() - timedelta(days = 2)
>>> data = self.get_history("light.office_lamp", start_time = start_time)
```

Get device state from yesterday and walk 5 days back.

```
>>> import datetime
>>> from datetime import timedelta
>>> end_time = datetime.datetime.now() - timedelta(days = 1)
>>> data = self.get_history(end_time = end_time, days = 5)
```

## 9.6 See More

Read the AppDaemon API Reference to learn other inherited helper functions that can be used by Hass applications.

# MQTT API Reference

A list of API calls and information specific to the MQTT plugin.

## 10.1 App Creation

To create apps based on just the MQTT API, use some code like the following:

```python
import mqttapi as mqtt

class MyApp(mqtt.Mqtt):

    def initialize(self):
```

## 10.2 Making Calls to MQTT

The MQTT Plugin uses the inherited `call_service()` helper function the AppDaemon API, to carry out service calls from within an AppDaemon app. See the documentation of this function here for a detailed description.

Th function `call_service()` allows the app to carry out one of the following services:

- `Publish`
- `Subscribe`
- `Unsubscribe`

By simply specifying within the function what is to be done. It uses configuration specified in the plugin configuration which simplifies the call within the app significantly. Different brokers can be accessed within an App, as long as they are all declared when the plugins are configured, and using the `namespace` parameter. See the section on namespaces for a detailed description.

### 10.2.1 Examples

```
# if wanting to publish data to a broker
self.call_service("publish", topic = "homeassistant/bedroom/light", payload = "ON")
# if wanting to unsubscribe a topic from a broker in a different namespace
self.call_service("unsubscribe", topic = "homeassistant/bedroom/light", namespace =
→"mqtt2")
```

The MQTT API also provides 3 convenience functions to make calling of specific functions easier and more readable. These are documented in the following section.

## 10.3 Reference

## 10.4 Services

**mqtt_subscribe** (*self*, *topic*, *\*\*kwargs*)

Subscribes to a MQTT topic.

This helper function used for subscribing to a topic on a broker, from within an AppDaemon App.

This allows the apps to now access events from that topic, in realtime. So outside the initial configuration at plugin config, this allows access to other topics while the apps runs. It should be noted that if AppDaemon was to reload, the topics subscribed via this function will not be available by default. On those declared at the plugin config will always be available. It uses configuration specified in the plugin configuration which simplifies the call within the app significantly.

Different brokers can be accessed within an app, as long as they are all declared when the plugins are configured, and using the `namespace` parameter.

> **Parameters**
>
> - **topic** (*str*) – The topic to be subscribed to on the broker (e.g., `homeassistant/bedroom/light`).
> - **\*\*kwargs** (*optional*) – Zero or more keyword arguments.
>
> **Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns** None.

#### Examples

```
>>> self.mqtt_subscribe("homeassistant/bedroom/light")
```

**mqtt_unsubscribe** (*self*, *topic*, *\*\*kwargs*)

Unsubscribes from a MQTT topic.

A helper function used to unsubscribe from a topic on a broker, from within an AppDaemon app.

This denies the Apps access events from that topic, in realtime. It is possible to unsubscribe from topics, even if they were part of the topics in the plugin config; but it is not possible to unsubscribe #. It should also be noted that if AppDaemon was to reload, the topics unsubscribed via this function will be available if they were configured with the plugin by default. It uses configuration specified in the plugin configuration which simplifies the call within the app significantly.

---

Different brokers can be accessed within an app, as long as they are all declared when the plugins are configured, and using the `namespace` parameter.

> **Parameters**
>
> - **`topic`** (`str`) – The topic to be unsubscribed from on the broker (e.g., `homeassistant/bedroom/light`).
>
> - **`**kwargs`** (`optional`) – Zero or more keyword arguments.
>
> **Keyword Arguments** **`namespace`** (`str, optional`) – Namespace to use for the call. See the section on [namespaces](#) for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns** None.

### Examples

```
>>> self.mqtt_unsubscribe("homeassistant/bedroom/light")
```

**`mqtt_publish`** (*self*, *topic*, *payload=None*, *\*\*kwargs*)

Publishes a message to a MQTT broker.

This helper function used for publishing a MQTT message to a broker, from within an AppDaemon app. It uses configuration specified in the plugin configuration which simplifies the call within the App significantly.

Different brokers can be accessed within an app, as long as they are all declared when the plugins are configured, and using the `namespace` parameter.

> **Parameters**
>
> - **`topic`** (`str`) – topic the payload is to be sent to on the broker (e.g., `homeassistant/bedroom/light`).
>
> - **`payload`** – data that is to be sent to on the broker (e.g., `'ON'`).
>
> - **`**kwargs`** (`optional`) – Zero or more keyword arguments.
>
> **Keyword Arguments**
>
> - **`qos`** (`int, optional`) – The Quality of Service (QOS) that is to be used when sending the data to the broker. This is has to be an integer (Default value: `0`).
>
> - **`retain`** (`bool, optional`) – This flag is used to specify if the broker is to retain the payload or not (Default value: `False`).
>
> - **`namespace`** (`str, optional`) – Namespace to use for the call. See the section on [namespaces](#) for a detailed description. In most cases it is safe to ignore this parameter.
>
> **Returns** None.

Examples:

Send data to the default HA broker.

```
>>> self.mqtt_publish("homeassistant/bedroom/light", "ON")
```

Send data to a different broker.

```
>>> self.mqtt_publish("homeassistant/living_room/light", "ON", qos = 0, retain =
→True, namespace = "mqtt2")
```

**is_client_connected**(*self*, *\*\*kwargs*)

>Returns `TRUE` if the MQTT plugin is connected to its broker, `FALSE` otherwise.

>This a helper function used to check or confirm within an app if the plugin is connected to its broker. This can be useful, if it is necessary to be certain the client is connected, so if not the app can internally store the data in a queue, and wait for connection before sending the data.

>Different brokers can be accessed within an app, as long as they are all declared when the plugins are configured, and using the `namespace` parameter.

>>**Parameters `**kwargs`** (`optional`) – Zero or more keyword arguments.

>>**Keyword Arguments `namespace`** (`str, optional`) – Namespace to use for the call. See the section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

>>**Returns** None.

### Examples

Check if client is connected, and send data. >>> if self.clientConnected(): >>> self.mqtt_publish(topic, payload)

Check if client is connected in mqtt2 namespace, and send data.

```
>>> if self.clientConnected(namespace = 'mqtt2'):
>>>     self.mqtt_publish(topic, payload, namespace = 'mqtt2')
```

## 10.5 Events

**listen_event**(*self*, *callback*, *event=None*, *\*\*kwargs*)

>Listens for changes within the MQTT plugin.

>Unlike other plugins, MQTT does not keep state. All MQTT messages will have an event which is set to `MQTT_MESSAGE` by default. This can be changed to whatever that is required in the plugin configuration.

>>**Parameters**

>>- **callback** – Function to be invoked when the requested event occurs. It must conform to the standard Event Callback format documented Here.

>>- **event** – Name of the event to subscribe to. Can be the declared `event_name` parameter as specified in the plugin configuration. If no event is specified, `listen_event()` will subscribe to all MQTT events within the app's functional namespace.

>>- **`**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 specific topic or wildcard can be listened to, instead of listening to all topics subscribed to. For example, if data is sent to a subscribed topic, it will generate an event as specified in the config; if we want to listen to a specific topic or wildcard, `topic` or `wildcard` can be passed in, and used to filter the callback by supplying them as keyword arguments. If you include keyword values, the values supplied to the ``listen_event()``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.

> **Keyword Arguments namespace** (*str, optional*) – Namespace to use for the call. See the
> section on namespaces for a detailed description. In most cases it is safe to ignore this parameter.

> **Returns** A handle that can be used to cancel the callback.

### Examples

Listen all events.

```
>>> self.listen_event(self.mqtt_message_received_event, "MQTT_MESSAGE")
```

Listen events for a specific subscribed topic.

```
>>> self.listen_event(self.mqtt_message_received_event, "MQTT_MESSAGE", topic =
→'homeassistant/bedroom/light')
```

Listen events for a specific subscribed high level topic.

```
>>> self.listen_event(self.mqtt_message_received_event, "MQTT_MESSAGE", wildcard␣
→= 'homeassistant/#')
```

Listen plugin's *disconnected* events from the broker.

```
>>> self.listen_event(self.mqtt_message_received_event, "MQTT_MESSAGE", state =
→'Disconnected', topic = None)
```

Listen plugin's' *connected* events from the broker.

```
>>> self.listen_event(self.mqtt_message_received_event, "MQTT_MESSAGE", state =
→'Connected', topic = None)
```

### Notes

At this point, it is not possible to use single level wildcard like using `homeassistant/+/light` instead of
`homeassistant/bedroom/light`. This could be added later, if need be.

## 10.6 MQTT Config

Developers can get the MQTT configuration data (i.e., client_id or username) using the helper function
`get_plugin_config()` inherited from the AppDaemon API. See the documentation of this function here for
a detailed description.

## 10.7 See More

Read the AppDaemon API Reference to learn other inherited helper functions that can be used by Hass applications.

# Dashboard Install and Configuration

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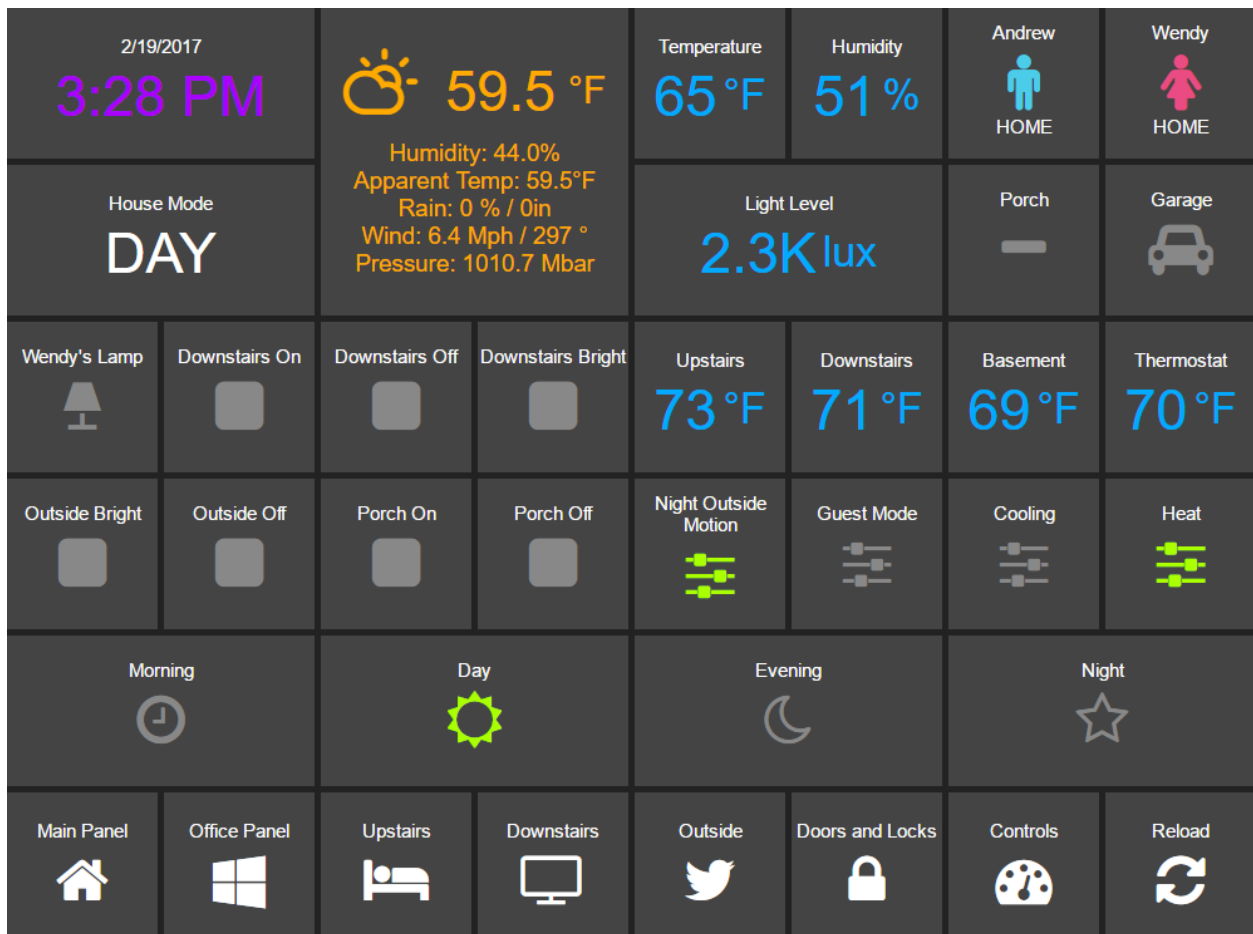
## 11.1 Installation and Configuration

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 `hadashboard` directive to the `appdaemon.yaml` file. For instance:

```
appdaemon:
    ...
hadashboard:
```

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 `dashboard_dir` directive. For example:

```
hadashboard:
    dashboard_dir: /etc/appdaemon/dashboards
```

Once initial configuration is complete, 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 dashboard, 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 dashboard 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 dashboards 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:
    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 `access_log` directives in the `logs` section. For example:

```
logs:
  access_log:
    filename: /export/pegasus/hass/appdaemon_test/logs/access.log
```

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

```
hadashboard:
  compile_on_start: 1
```

This should not be necessary but may on occasion be required after an upgrade to pickup changes. This is now the default if not otherwise specified.

## 11.2 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 dashboard called main with the obsidian skin:

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

# Dashboard Creation
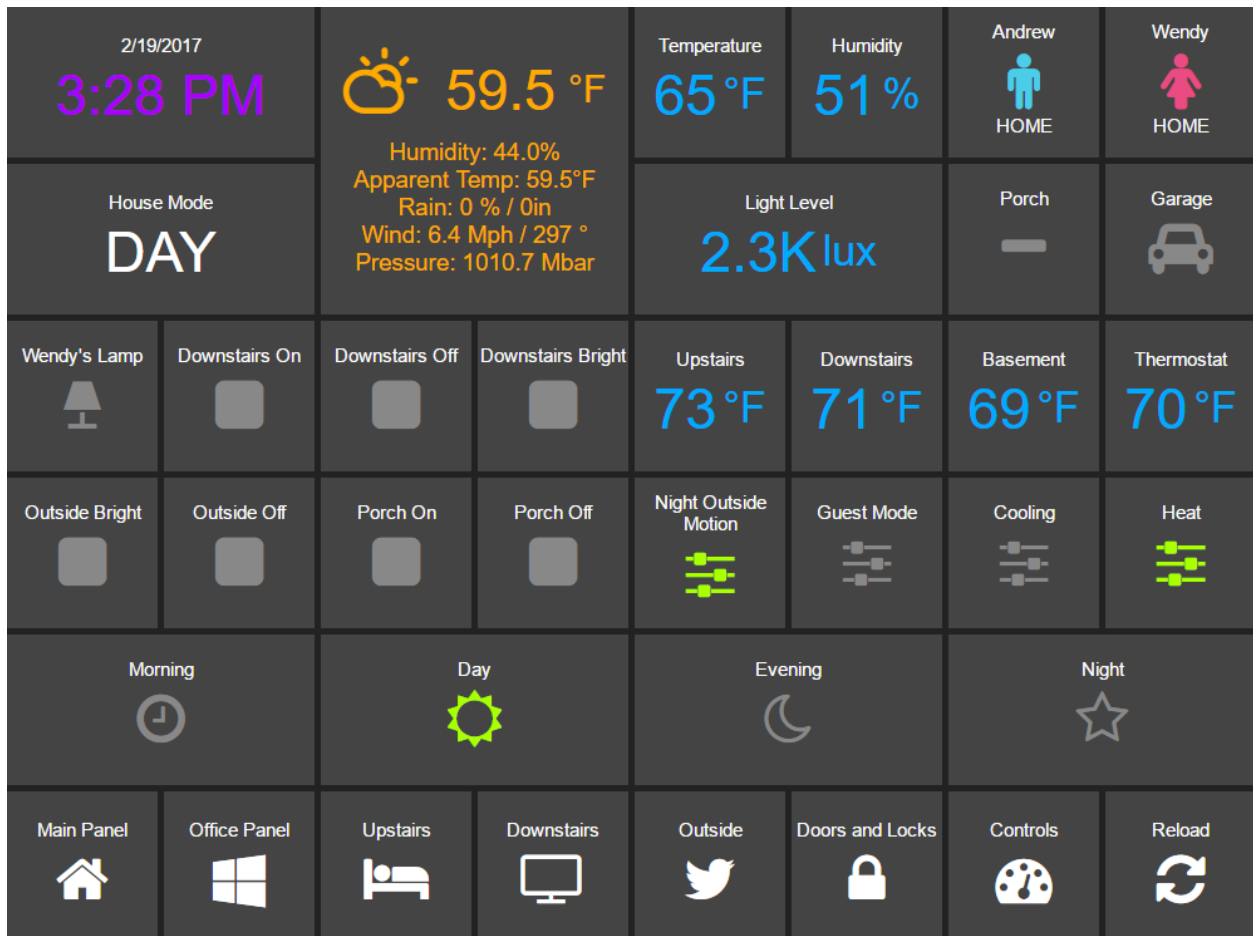


Fig. 1: UI

# 12.1 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.

## 12.1.1 Main Settings

A top-level dashboard will usually have one of several 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
    namespace: default
    devices:
      media_player:
        step: 5
```

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 tablet's 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 dashboard will have.

- `scalable` - if set to `False` this parameter will disable resizing and double-tap zooming on iOS devices, default is not to disable zooming.

- `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 overridden 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 namespace parameter will be explained further in the namespace section of this document. Within the `global` parameters it is also possible to set parameters at the device level by including a `device` entry (see above for an example). Under device you can add an entry for any widget type, then under

that, list global parameters that will be applied to just that widget type. For instance, in the example above, the default step size for all media players is set to 5% rather than the default 10%.

The very simplest dashboard needs a layout so it can understand where to place the widgets. We use a `layout` directive to tell HADashboard 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 referring directly to native Home Assistant entities. From this, HADashboard is able to figure out the right widget type and grab its friendly name and add it to the dashboard. 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 layout 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. HADashboard will do its 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.

## 12.1.2 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 define the widget along with its associated parameters formally.

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 attributes 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:

```
andrew_presence:
    widget_type: device_tracker
    title: Andrew
    device: andrews_iphone

clock:
    widget_type: clock

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

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

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

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

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
```

(continues on next page)

```
    entity: sensor.side_humidity_corrected

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

wendy_presence:
    widget_type: device_tracker
    title: Wendy
    device: wendys_iphone
```

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 that 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.

## 12.2 Advanced Dashboard Definition

When you get to the point where you have multiple dashboards, you may want to take a more modular approach. For example, 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 its 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 wherein 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 whereas 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 submodules (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: fas-car
    icon_off: fas-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 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 subfiles:

- Sub files can include other subfiles to a maximum depth of 10 - please avoid circular references!

---

> • When layout information is included in a subfile, 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 on 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
```

## 12.3 Widget Customization

Widgets allow customization using arbitrary 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 HADashboards 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.

## 12.4 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
- icon
- 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 to be 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.

## 12.5 Titles

Each widget could have custom text for title a title2. You can use the option to force the widget to use text from entity friendly name attribute.

- `title_is_friendly_name` - set title as entity friendly name if exists
- `title2_is_friendly_name` - set title2 as entity friendly name if exists

---

Example:

```
title_is_friendly_name: 1
title2_is_friendly_name: 1
```

## 12.6  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 `mdi-` for Material Design, and the appropriate style prefix for Font Awesome Icons.

```
icon_on: fas-bell
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.

## 12.7  External Commands

The dashboard can accept commands 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 its 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
    sticky: 0
```

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

### 12.7.1  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. `sticky` - whether or not to return to the original dashboard after it has been clicked on. The default behavior (`sticky=0`) is to remain on the new dashboard if clicked and return to the original otherwise. With `sticky=` `, clicking the dashboard will extend the amount of time, but it will return to the original dashboard after a period of inactivity equal to `timeout`.

## 12.8 Namespaces

For a full explanation of namespaces see the `Writing AppDaemon Apps` Section of the guide. Namespaces may be ignored in HADashboard if only one plugin is in use.

If multiple namespaces are in use, HADashboard is able to specify either at the dashboard level or the widget level which namespace to use. This is achieved by use of the `namespace` parameter. This parameter may be specified for each individual widget if desired. If it is specified as one of the global parameters, it will apply to all widgets but may be overridden for individual widgets. If not specified as a global parameter, the default namespace will be used for any widgets that do not override it. For example:

```
##
## 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
    # Not setting namespace here so the default namespace is used

# Clock has no namespace
clock:
    widget_type: clock

# side_temperature doesn't specify a namespace so will use the default
# If we specified a different namespace in the global options it would use that
→instead
side_temperature:
    widget_type: sensor
    title: Temperature
    units: "&deg;F"
    precision: 0
    entity: sensor.side_temp_corrected

# side_humidity overrides the default and uses the hass2 namespace
# It will use hass2 regardless of any global setting
side_humidity:
```

(continues on next page)

```
namespace: hass2
widget_type: sensor
title: Humidity
units: "%"
precision: 0
entity: sensor.side_humidity_corrected
```

One caveat to namespaces is that the RSS widget always works with the admin namespace - since the RSS feeds are supplied by AppDaemon itself, and not one of the plugins.

## 12.9 Widget Reference

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

### 12.9.1 alarm
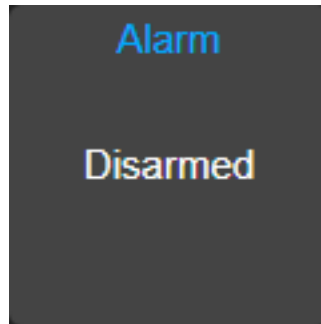


A widget to report on the state of an alarm and allow code entry

#### 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`

### 12.9.2 binary_sensor
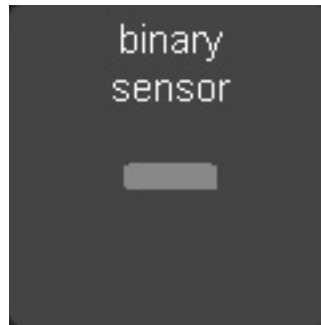


A widget to monitor a binary_sensor

#### 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`

#### Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`
- `state_text_style`

### 12.9.3 camera

A widget to display a refreshing camera image on the dashboard.

**Mandatory arguments:**

- `entity` the entity_id of the camera
- `base_url` the URL to your Home Assistant install. If you wish to access your dashboard from an external network, then this needs to be the external URL to Home Assistant.

**Optional Arguments:**

- `refresh` (seconds) - if set, the camera image will refresh every interval. Use 0 to indicate no refresh at all. Default is 10 seconds for non-streaming cameras and 0 for streaming cameras.
- `stream` - If set to *on* live data will be streamed to the dashboard instead of needing to be refreshed.

**Style Arguments:**

- `widget_style`
- `title_style`

## 12.9.4  climate

A widget to monitor and control a climate entity

**Mandatory arguments:**

- `entity` - the entity_id of the climate entity
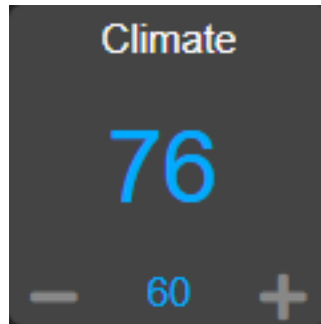
**Optional Arguments:**

- `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
- `precision` - the number of digits to display after the decimal point

**Style Arguments:**

- `widget_style`
- `icon_up`
- `icon_down`
- `title_style`
- `title2_style`
- `level_style`
- `level2_style`
- `unit_style`
- `unit2_style`
- `level_up_style`
- `level_down_style`

### 12.9.5 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

- `date_format_country` - Format the clock in the style of a specific country. This can take a simple value like `us` or more complex parameters as described here.

- `date_format_options` - if using `date_format_country` you can also add additional options for formatting as described here.. For example:

```
clock:
    widget_type: clock
    date_format_country: "ro"
    date_format_options:
      weekday: "short"
      day: "numeric"
      month: "numeric"
```

**Style Arguments:**

- `widget_style`

- `time_style`

- `date_style`

## 12.9.6 cover



images/cover.png

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
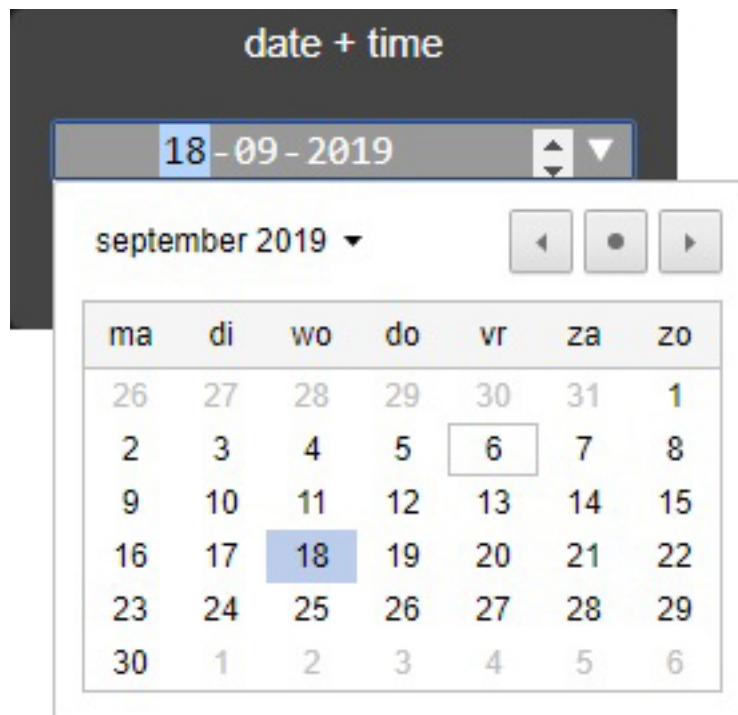
---

**Optional Arguments:**

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

**Style Arguments:**

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

## 12.9.7 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.

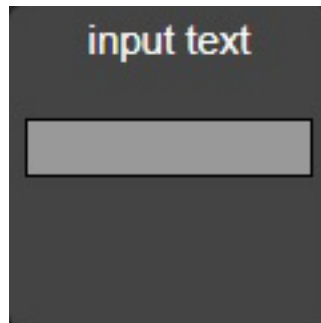**Optional Arguments:**

- `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`

## 12.9.8 entitypicture



A widget to display entity picture

### Mandatory arguments:

- `entity` - the entity to display entity_picture attribute

**Optional Arguments:**

- `title` - the title displayed on the tile.

- `base_url` - URL to prepend before content of entity_picture.

Example:

```
entitypicture1:
    widget_type: entitypicture
    title: Weather by YR
    entity: sensor.yr_symbol
```

Example showing artwork of just playing an album on media player: (tested with Google Home)

```
entitypicture2:
    widget_type: entitypicture
    entity: media_player.bedroom
    base_url: https://my_domain.duckdns.org:8123
    image_style: "top: 0; bottom: 0; left: 0; right: 0;"
```

**Style Arguments:**

- `widget_style`

- `title_style`

- `image_style`

## 12.9.9 gauge
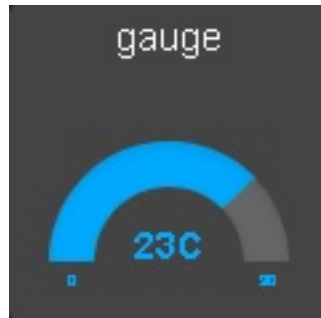


A widget to report on numeric values for sensors in Home Assistant in a gauge format.

**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.

### 12.9.10  group
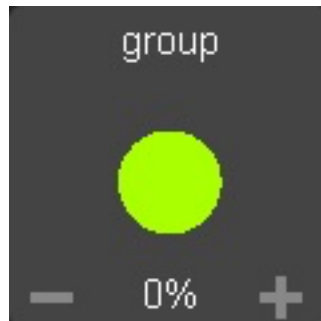


A widget to monitor and control a group of lights

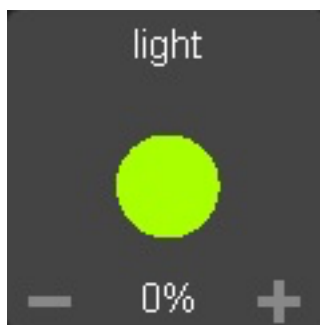**Mandatory arguments:**

- `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.

**Style 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`

## 12.9.11 icon



A widget to monitor the state of an entity and display a different icon and style for each listed state, and is configured in a similar manner to the following:

```
icon:
  title: icon
  widget_type: icon
  entity: binary_sensor.basement_door_sensor
  state_text: 1
  icons:
    "active":
      icon: fas-glass
      style: "color: green"
```

(continues on next page)

```yaml
    "inactive":
      icon: fas-repeat
      style: "color: blue"
    "idle":
      icon: fas-frown
      style: "color: red"
    "default":
      icon: fas-rocket
      style: "color: cyan"
```

The icons list is mandatory, and each entry must contain both an icon and a style entry. It is recommended that quotes are used around the state names, as without these, YAML will translate states like `on` and `off` to `true` and `false`

The default entry icon and style will be used if the state doesn't match any in the list - meaning that it is not necessary to define all states if only 1 or 2 actually matter.

### Mandatory arguments:

- `entity` - the entity_id of the binary_sensor
- `icons` - a list of icons and styles to be applied for various states.

### Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `state_text`
- `state_map`

### Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `state_text_style`

## 12.9.12 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.

---
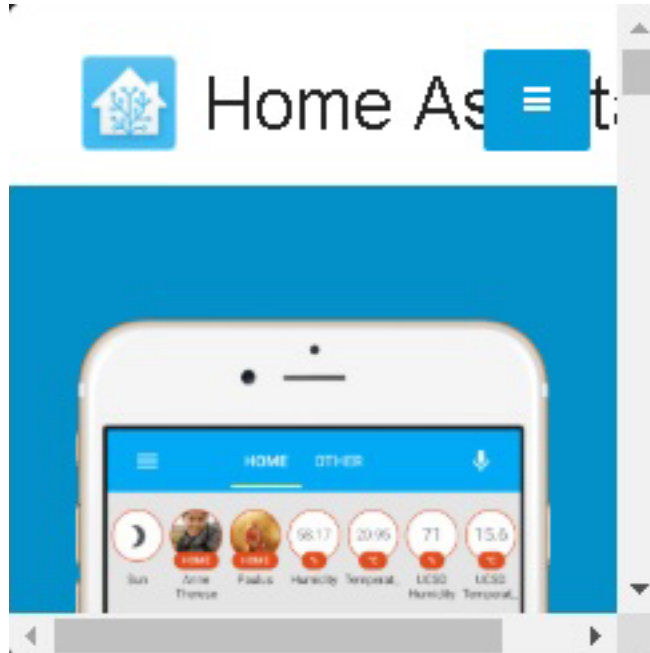
**Optional Arguments:**

- `title` - the title displayed on the tile

- `refresh` - (seconds) if set, the iframe widget will progress down its 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-sofas-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: ""
```

(continues on next page)

```
   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
```

**Style Arguments:**

- `widget_style`

- `title_style`

### 12.9.13 input_boolean
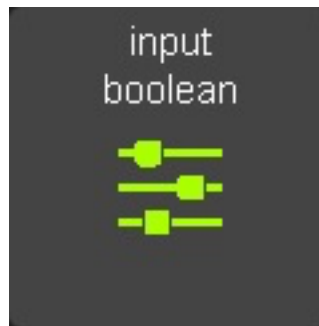


A widget to monitor and activate an input_boolean

**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`

**Style Arguments:**

- `icon_on`

- `icon_off`

- `widget_style`

- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

## 12.9.14 input_datetime



A widget to monitor and control an input_datetime

### Mandatory arguments:

- `entity` - the entity_id of the input_datetime

### Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text

### Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `container_style`

- `date_style`
- `time_style`

## 12.9.15  input_number
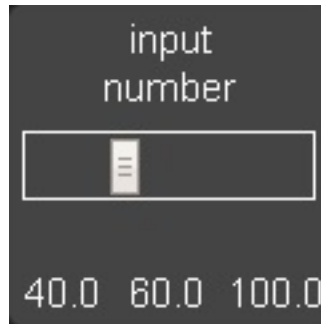


A widget to monitor and control an input_number

### Mandatory arguments:

- `entity` - the entity_id of the input_number

### Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `units` - the unit symbol to be displayed
- `use_comma` - if set to one, a comma will be used as the decimal separator

### Style Arguments:

- `title_style`
- `title2_style`
- `minvalue_style`
- `maxvalue_style`
- `value_style`
- `slider_style`
- `slidercontainer_style`
- `widget_style`

## 12.9.16  input_select

A widget to display and select values from an input_select entity in Home Assistant.

---

**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

**Style Arguments:**

- `widget_style`
- `title_style`
- `title2_style`
- `select_style`
- `selectcontainer_style`

## 12.9.17 input_slider



An alternate widget to monitor and control an input number, using plus and minus buttons instead of a slider.

**Mandatory arguments:**

- `entity` - the entity_id of the input_number

**Optional Arguments:**

- `title` - the title displayed on the tile
- `title2` - a second line of title text
- `units` - the unit symbol to be displayed
- `use_comma` - if set to one, a comma will be used as the decimal separator

**Style Arguments:**

- `title_style`
- `title2_style`
- `minvalue_style`
- `maxvalue_style`
- `value_style`
- `slider_style`
- `slidercontainer_style`
- `widget_style`

## 12.9.18 input_text



A widget to monitor and control an input_text

**Mandatory arguments:**

- `entity` - the entity_id of the input_text

**Optional Arguments:**

- `title` - the title displayed on the tile
- `title2` - a second line of title text

**Style Arguments:**

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

## 12.9.19 javascript



A widget to run an arbitrary JavaScript command.

### Mandatory arguments:

- `command` - the JavaScript command to be run.

e.g.

### Optional Arguments:

- `title` - the title displayed on the tile
- `title2` - a second line of title text

### Style Arguments:

- `icon_active`
- `icon_inactive`
- `widget_style`
- `title_style`
- `title2_style`
- `icon_active_style`
- `icon_inactive_style`

## 12.9.20 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

### Style Arguments:

- `widget_style`
- `title_style`
- `title2_style`
- `value_style` - changes the style from the text

## 12.9.21 light



A widget to monitor and control 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 syntax as in Home Assistant scenes. See below for examples.

```
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
```

**Style 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`

## 12.9.22  lock
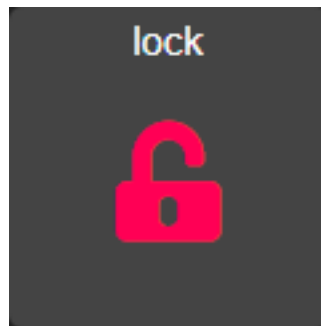


A widget to monitor and activate a lock

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, whereas 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`

### Style Arguments:

- `icon_on`
- `icon_off`
- `widget_style`

- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

### 12.9.23 london_underground



```
images/london_underground.png
```
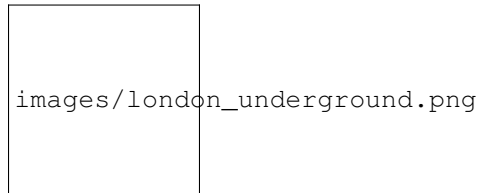
A widget to report on the status of a London Underground line and provide the reason for delays if there are any. Requires the London Underground sensor to be configured in Home Assistant. This widget is designed to be a 2x2 tile.

It is recommended to update the background style to reflect the color of the underground line. An example would be as follows:

```
widget_style: "background-color: #0098D4"
```

The colors of the various lines are: - Bakerloo: #B36305 - Central: #E32017 - Circle: #FFD300 - District: #00782A - DLR: #00A4A7 - Hammersmith & City: #F3A9BB - Jubilee: #A0A5A9 - London Overground: #EE7C0E - Metropolitan: #9B0056 - Northern: #000000 - Piccadilly: #003688 - Victoria: #0098D4 - Waterloo & City: #95CDBA

For smaller dashboards the Description text can be too long to fit in the widget properly. In that case, hide the text as follows:

```
state_text_style: "display: none"
```

**Mandatory arguments:**

- `entity` - the entity to be monitored

**Optional Arguments:**

- `title` - the title displayed on the tile

**Style Arguments:**

- `state_text_style`
- `text_style`
- `title_style`
- `widget_style`
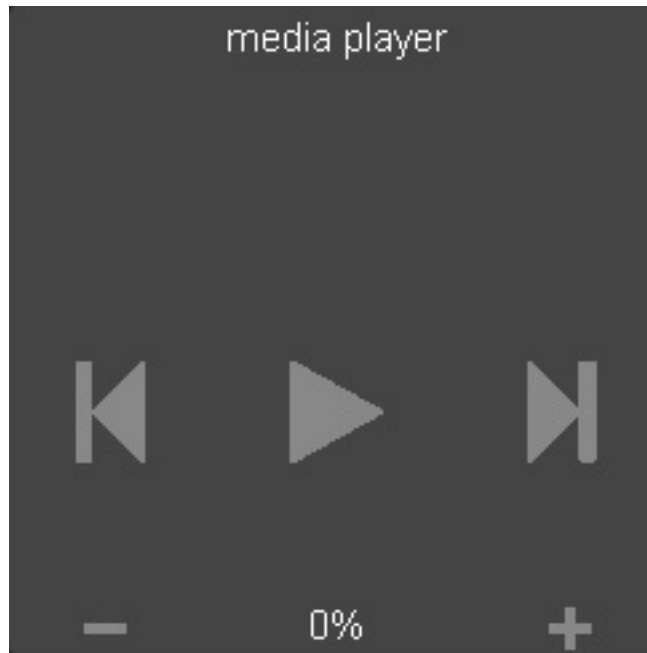
### 12.9.24 media_player

A widget to monitor and control a media player
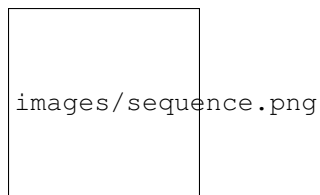
**Mandatory arguments:**

- `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%)

**Style 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`

### 12.9.25 mode
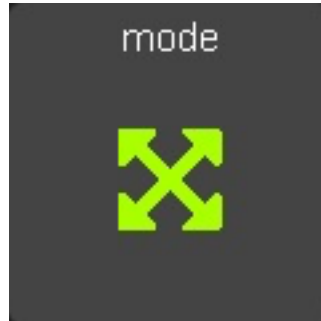


A widget to track the state of an `input_select` by showing active when it is set to a specific value. Also allows scripts to be run when activated.

#### 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`

## 12.9.26 navigate



A widget to navigate to a new URL, intended to be used for switching between dashboards.

### Mandatory arguments:

None, but either `url` or `dashboard` must be specified.

### Optional Arguments:

- `url` - a url to navigate to. Use a full URL including the *http://* or *https://* 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 arbitrary 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.
- `sticky` - whether or not to return to the original dashboard after it has been clicked on. The default behavior (`sticky=0`) is to remain on the new dashboard if clicked and return to the original otherwise. With `sticky=1`, clicking the dashboard will extend the amount of time, but it will return to the original dashboard after a period of inactivity equal to `timeout`.

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
```

**Style Arguments:**

- `icon_active`

- `icon_inactive`

- `widget_style`

- `title_style`

- `title2_style`

- `icon_active_style`

- `icon_inactive_style`

### 12.9.27 person
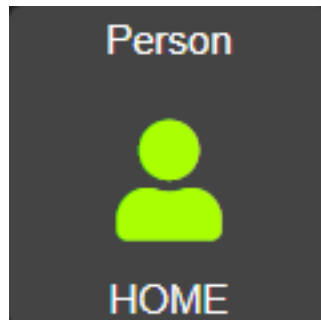


A Widget that reports on the status of a person. It can also be optionally be used to toggle the status between "home" and "not_home".

**Mandatory Arguments:**

- `entity` - entity of the person, e.g. `person.andrew`.

**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 person 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: person
  title: Wendy
  title2: Mapped
  entity: person.wendy
  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`

## 12.9.28 radial



A widget to display a numeric value as a gauge

**Mandatory Arguments:**

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

- `settings` - a list if values describing the gauge with the following entries:

- title - title of the guage

- minValue - minimum value to display

- maxValue - maximum value to display

- majorTicks - Where to mark major values, a list

- highlights - color ranges, a list

See the example below:

```
your_radial:
  widget_type: radial
  entity: sensor.your_sensor
  settings:
    title: any title
    minValue: 0
    maxValue: 100
    majorTicks: [0,20,40,60,80,100]
    highlights: [{'from': 0, 'to': 18, 'color': 'rgba(0,0, 255, .3)'},{'from': 25, 'to
→': 100, 'color': 'rgba(255, 0, 0, .3)'}]
```

**Optional Arguments:**

None

**Style Arguments:**

None

## 12.9.29 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

**Style Arguments:**

- `icon_active`
- `icon_inactive`
- `widget_style`
- `title_style`
- `title2_style`
- `icon_active_style`
- `icon_inactive_style`

### 12.9.30 rss
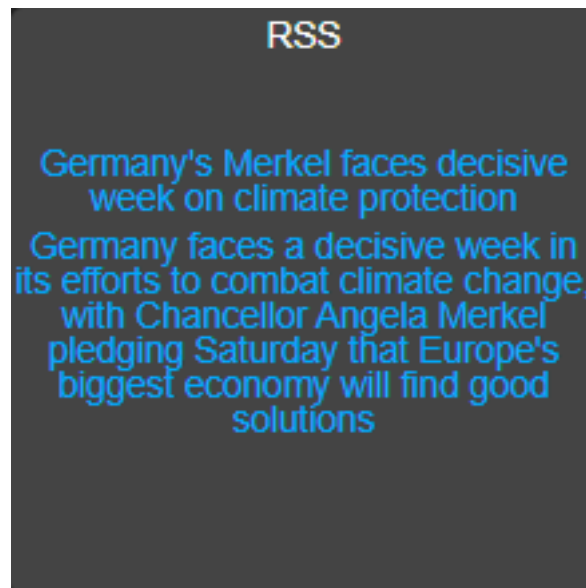


A widget to display an RSS feed.

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

```
hadashboard:

  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. This must be an arbitrary name prepended by `rss.` - e.g., `rss.cnn_news`
- `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.

### Mandatory Arguments:

- `entity` - the name of the configured feed - this must match the `target_name` full target name configured in the AppDaemon configuration e.g. *rss.cnn_news*
- `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.
- `show_description` - if set to `1` the widget will show a short description of the story as well as the title. Default is `0`

### Style Arguments:

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

## 12.9.31 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`

## 12.9.32  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`
- `enabled` - if set to 0 the switch cant be pressed but only shows status
- `momentary` - after the set amount of milliseconds the old state returns (momentary button)

**Style Arguments:**

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

## 12.9.33 scene



A widget to run a sequence

**Mandatory arguments:**

- `entity` - the entity_id of the sequence e.g. `sequence.office_on`

**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`

### 12.9.34 sensor
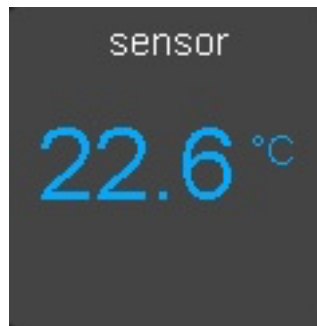


A widget to report on values for any sensor in Home Assistant

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. To display an attribute of a sensor rather than the state itself add the attribute to the end of the sensor name. For example, to display the description of the sensor.dark_sky_summary sensor you would use the following entity definition: "sensor.dark_sky_summary.Description".

Note that you can define a sub_entity to be an attribute of the entity using the entity_to_sub_entity_attribute argument, or an entity as an attribute of the sub_entity using the sub_entity_to_entity_attribute.

**Mandatory Arguments:**

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

Or

- `sub_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
- `entity_to_sub_entity_attribute` - the attribute of the entity to use as the sub_entity
- `sub_entity_to_entity_attribute` - the attribute of the sub_entity to use as the entity

**Style Arguments:**

- `widget_style`
- `title_style`
- `title2_style`
- `value_style`
- `text_style`
- `unit_style`
- `container_style`
- `state_text_style` (used for styling of `sub_entity`)

## 12.9.35 switch
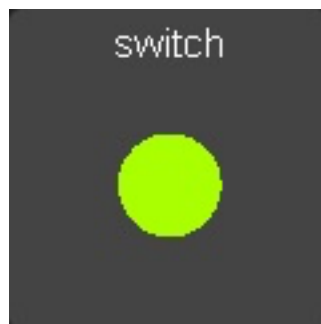


A widget to monitor and activate a switch

**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`
- `enabled` - if set to 0 the switch cant be pressed but only shows status
- `momentary` - after the set amount of milliseconds the old state returns (momentary button)

**Style Arguments:**

- `icon_on`
- `icon_off`
- `widget_style`
- `icon_style_active`
- `icon_style_inactive`
- `title_style`
- `title2_style`

## 12.9.36 temperature



A widget to report display a temperature using a thermometer style view

**Mandatory Arguments:**

- `entity` - the entity_id of the alarm to be monitored
- `settings` - a list if values describing the thermometer with the following entries:
- minValue - minimum value to display
- maxValue - maximum value to display
- width - width of the widget, set this to the same width as your cell size or less
- height - height of the widget, set this to the same height as your cell size or less
- majorTicks - Where to mark major values, a list
- highlights - color ranges, a list

See the example below:

```
your_temperature:
  widget_type: temperature
  entity: sensor.your_sensor
  settings:
    minValue: 15
    maxValue: 30
    width: 120
    height: 120
    majorTicks: [15,20,25,30]
    highlights: [{'from': 15, 'to': 18, 'color': 'rgba(0,0, 255, .3)'},{'from': 24,
→'to': 30, 'color': 'rgba(255, 0, 0, .3)'}]
```

**Optional Arguments:**

None

**Style Arguments:**

None

### 12.9.37 weather



Up to date weather reports. By default, it's configured to work with dark sky sensor. To use all the features, you need to add these sensors to monitored_conditions:

- temperature
- apparent_temperature
- temperature_min
- temperature_max

- humidity

- precip_probability

- precip_intensity

- precip_type

- wind_speed

- wind_bearing

- pressure

- icon

To have the forecast displayed set `show_forecast` to 1. For it to work you additionally, need to add the forecast option in dark_sky Home Assistant configuration.

```
forecast:
  - 1
```

### Mandatory arguments:

None

### Optional Arguments:

- `title`

- `show_forecast` - show the forecast

- `prefer_icons` - use icons instead of text

- `forecast_title` - title of the forecast if enabled

- `sensors` - list of sensors used by the widget

You can change the entities used by the widget by overwriting their values in the `sensors` key in configuration.

Example with default values:

```
sample_weather:
  widget_type: weather
  title: Today
  show_forecast: 1
  prefer_icons: 1
  forecast_title: Tomorrow
  sensors:
    icon: sensor.dark_sky_icon
    temperature: sensor.dark_sky_temperature
    apparent_temperature: sensor.dark_sky_apparent_temperature
    humidity: sensor.dark_sky_humidity
    precip_probability: sensor.dark_sky_precip_probability
    precip_intensity: sensor.dark_sky_precip_intensity
    precip_type: sensor.dark_sky_precip
    pressure: sensor.dark_sky_pressure
    wind_speed: sensor.dark_sky_wind_speed
    wind_bearing: sensor.dark_sky_wind_bearing
    forecast_icon: sensor.dark_sky_icon_1
```

(continues on next page)

```
    forecast_temperature_min: sensor.dark_sky_daily_low_temperature_1
    forecast_temperature_max: sensor.dark_sky_daily_high_temperature_1
    forecast_precip_probability: sensor.dark_sky_precip_probability_1
    forecast_precip_type: sensor.dark_sky_precip_1
```

**Style Arguments:**

- `widget_style`

- `main_style`

- `unit_style`

- `sub_style`

- `sub_unit_style`

- `title_style`

### 12.9.38 weather_summary
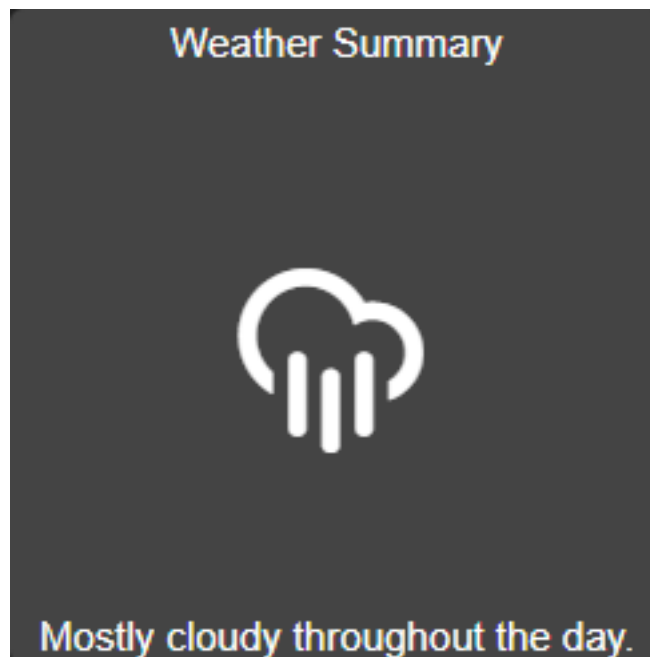


An icon and summary reflecting the weather forecast. Requires dark sky to be configured in Home Assistant and expects to be used with one of the following sensors:

- sensor.dark_sky_daily_summary

- sensor.dark_sky_hourly_summary

- sensor.dark_sky_summary

**Mandatory arguments:**

- `entity` - the entity to be monitored

**Optional Arguments:**

- `title` - the title displayed on the tile

**Style Arguments:**

- `state_text_style`

- `text_style`

- `title_style`

- `widget_style`

## 12.10 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 widget to switch between dashboards and will stay in force until another skin or no skin is specified.

HADashboard 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`

## 12.11 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 the 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 override 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: fas-circle
light_icon_off: fas-circle-thin
light_icon_up: fas-plus
light_icon_down: fas-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 existing file and edit to suit your needs.

## 12.12 Example Dashboards

Some example dashboards are available in the AppDaemon repository:

Dashboards

## 12.13 A Note on Font Awesome Upgrade

As of AppDaemon 3.0.2, Font Awesome icons have been upgraded from version 4 to version 5. FA Introduced a lot of breaking changes with this upgrade. While all of HADashboard's included skins have been updated to reflect this, any custom skins may need changes, as will any custom icons used within dashboard config files. FA have provided a table of changed icons here.

To ease the transition further, a legacy mode has been included in HADashboard. This is not enabled by default, but can be turned on by specifying the following in the hadashboard section of `appdaemon.cfg`:

```
fa4compatibility: 1
```

This is not intended as a permanent fix and may be removed at some point, but for now, this will enable existing skins and icons to work correctly, giving you an opportunity to work through your configurations and fix things.

While working through the upgrade, it is strongly advised that you clear your browser cache and force recompile all of your dashboards to flush out references to old icons. This can be done by manually removing the `compiled` subdirectory in `conf_dir`, specifying `recompile=1` in the arguments to the dashboard, or setting the hadashboard option `dash_compile_on_start` to `1`.

# 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.

## 13.1 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 about 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.

## 13.2 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 usual way.

## 13.3 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.

### 13.3.1 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.

### 13.3.2 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.

### 13.3.3 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.

### 13.3.4 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.

### 13.3.5 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 manipulated 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 in 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 the style when the button is pressed.

### 13.3.6 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 `static_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 to things like titles that remain static for the lifetime of the widget.

### 13.3.7 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: []
```

### 13.3.8 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.

## 13.4 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 purposes of this document, we will provide examples from the `baselight` Base Widget.

## 13.4.1 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 various tag types have styling suitable for some common elements although that can be overidden 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 programmatically 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 maintainable.

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 referred to later in the JavaScript

- The `<div>` 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.

## 13.4.2 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 ensuring 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;
```

(continues on next page)

```
}

.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;
}
```

### 13.4.3 Widget JavaScript Files

The JavaScript file is responsible for glueing all the pieces together:

- Registering callbacks for events
- Registering callbacks for touches
- Updating the fields, icons, styles as necessary

Let's 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 function 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 depending on the mechanism for calling into 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 recommended 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.

When subscribing to events that relate to value changes in a widget, such as for instance an input select being changed by a user, which we must propagate back to Home Assistant, there is an issue with race conditions if we subscribe to the normal *change* event. The *change* event will fire, and our *onChange* function may be called before the knockout binding has an opportunity to update itself, and we will see the old value. To handle this situation, a second type of event subscription is provided - we will subscribe to the knockout binding changing rather than the control itself. This is done in a similar way to the previous mechanism, the only difference is that instead of a *selector* parameter, we use an *observable* parameter which is the name of the binding you want to subscribe to. For instance:

```
{"observable": "selectedoption", "action": "change", "callback": self.onChange}
```

Both styles of callback may be used together.

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 when 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 other 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 - its 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` which should be set to a service that will turn the 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 into the widget.

Usually, HADashboard understands `args` values as a single string. If you need to use a service that expects to receive a list or a dictionary then you may use the special key `json_args` and set its value to a stringified json. For example, suppose you want to pass to the service a list called `colors`, then you could change the above code and include another check:

```
if ("my_json" in self.parameters)
{
   args["json_args"] =  JSON.stringify(self.parameters.my_json);
}
```

The corresponding widget configuration may include something like this:

```
my_json:
   colors:
     - red
     - blue
     - green
```

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 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
```

```javascript
    // 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)
    }
```

```javascript
// The OnStateUpdate function will be called when the specific entity
// receives a state update - its 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;
```

```javascript
        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 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)
```

```
            }
        }
}
```

## 13.5  A Note on Skinning

As you have seen, when creating a new widget, 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.

# Development

If you want to help with the development of AppDaemon all assistance is gratefully received! Here are a few things you can do to help.

## 14.1 Running a Dev Version

For the adventurous among you, it is possible to run the very latest dev code to get a preview of changes before they are released as part of a stable build. Be aware you use the dev branch at your own risk. Although we try to keep things consistent and functional, we can't guarantee that things won't break. However, feedback from brave souls running the dev branch is always gratefully received!

Also, note, that to run a dev version you should be using the *PIP install method*. Docker builds are created for dev too, but there is no hass.io support.

There are 2 different ways of installing via PIP. If we are running a beta, we will have a number of specific milestone builds. These will not install by default using the standard pip command line but can be installed if their exact version is given to the pip command:

```
$ pip3 install appdaemon==<specific beta version>
```

For non-beta builds, when the goal is just to run the latest dev code, or if you want a local version of the dev code without installing it over the top of the stable code, take the following steps:

### 14.1.1 Clone the Repository

First, we need to get a clean copy of the dev branch. To do this, create a new directory, and change into it. Run the following command to clone the dev branch of the AppDaemon repository:

```
$ git clone -b dev https://github.com/home-assistant/appdaemon.git
```

This will create a directory called appdaemon - this is your repository directory, and all commands will need to be run from inside it.

### 14.1.2 Run AppDaemon from the command line

Now that you have a local copy of the code, the next step is to run AppDaemon using that code.

As a first step, if you are using a Virtual Environment (VE) enable it. The best practice here is to use a VE specifically for the dev version. In some cases, it is possible that the dev branch may have updated dependencies that will be incompatible with the latest stable release, and may break it. In this process, you can find dependency issues, review `setup.py` for a list of required dependencies.

To run the cloned version of AppDaemon, make sure you are in the `appdaemon` subdirectory and run the following command:

```
$ python3 -m appdaemon -c <PATH To CONFIG DIRECTORY>
```

In most cases, it is possible to share config directories with other AppDaemon instances. However, you must be aware of apps that use new features as they will likely cause errors for the stable version. If you prefer, you can create an entirely new conf directory for your dev environment.

### 14.1.3 Install AppDamon via PIP (Optional)

Although the recommended way of running a dev build is to use the command line above, it is possible to install an AppDaemon dev build as a pip package. If you do so, it will replace your stable version, so only do this if you are confident with packages and VEs. However, if you use a specific VE for the dev build, this should not be an issue. Also, remember that if you do this, you will need to reinstall the package as an extra step every time you refresh the dev repository (see below).

To install the dev build as a package, change to the `appdaemon` directory and run the following command:

```
$ pip3 install .
```

### 14.1.4 Updating AppDaemon to the latest dev version

When the dev version has been updated, and you want to pull over the latest changes, run the following command from the `appdaemon` directory:

```
$ git pull
```

You can then immediately run the latest version with the command line above. If you are using pip, remember to rerun the `install` command using the `--upgrade flag`:

```
$ pip3 install --upgrade .
```

## 14.2 Pull Requests

If you see a way to improve on AppDaemon, We are pleased to receive Pull Requests. The official AppDaemon repository is here:

https://github.com/home-assistant/appdaemon

Please note, if documentation is required to make sense of the PR, the PR will not be accepted without it.

## 14.3 Documentation

Assistance with the docs is always welcome, whether its fixing typos and incorrect information or reorganizing and adding to the docs to make them more helpful. To work on the docs, submit a pull request with the changes, and I will review and merge them in the usual way. I use *readthedocs* to build and host the docs, and you can easily set up a preview of your edits as follows:

First, install sphinx.

```
$ pip3 install sphinx
```

Then *cd* to the *docs* subdirectory, where all the *rst* files are found, and run the following command:

```
$ sphinx-autobuild -H 0.0.0.0 . _build_html
```

Sphinx will take a minute or so to build the current version of the docs, and it will then be available on port 8000 (e.g., http://localhost:8080) of the machine hosting sphinx. As you make changes. Sphinx will automatically detect them and update the browser page in real-time. When you finish your editing, stop sphinx by typing ctrl-c.

# Internal Documentation

These notes are intended to assist anyone that wants to understand AppDaemon's internals better. Most modules are used from within the AppDaemon object, which is a centralized depository of configuration information and references to the other objects and subsystems within AppDaemon.

## 15.1 appdaemon object

## 15.2 admin

## 15.3 admin_loop

## 15.4 app_management

## 15.5 callbacks

## 15.6 dashboard

## 15.7 events

Module to handle all events within AppDaemon.

**class Events**(*ad: appdaemon.appdaemon.AppDaemon*)
> Encapsulate event handling.

> **add_event_callback**(*name*, *namespace*, *cb*, *event*, *\*\*kwargs*)
> > Adds a callback for an event which is called internally by apps.

> > **Parameters**

- **name** (*str*) – Name of the app.

- **namespace** (*str*) – Namespace of the event.

- **cb** – Callback function.

- **event** (*str*) – Name of the event.

- **\*\*kwargs** – List of values to filter on, and additional arguments to pass to the callback.

> **Returns** None or the reference to the callback handle.

**cancel_event_callback**(*name*, *handle*)
>     Cancels an event callback.

>     **Parameters**

- **name** (*str*) – Name of the app or module.

- **handle** – Previously supplied callback handle for the callback.

>     **Returns** None.

**fire_event**(*namespace*, *event*, *\*\*kwargs*)
>     Fires an event.

>     If the namespace does not have a plugin associated with it, the event will be fired locally. If a plugin is associated, the firing of the event will be delegated to the plugin, under the understanding that when the event is fired, the plugin will notify appdaemon that it occurred, usually via the system the plugin is communicating with.

>     **Parameters**

- **namespace** (*str*) – Namespace for the event to be fired in.

- **event** (*str*) – Name of the event.

- **\*\*kwargs** – Arguments to associate with the event.

>     **Returns** None.

**has_log_callback**(*name*)
>     Returns True if the app has a log callback, False otherwise.

>     Used to prevent callback loops. In the calling logic, if this function returns True the resulting logging event will be suppressed.

>     **Parameters** **name** (*str*) – Name of the app.

**info_event_callback**(*name*, *handle*)
>     Gets the information of an event callback.

>     **Parameters**

- **name** (*str*) – Name of the app or subsystem.

- **handle** – Previously supplied handle for the callback.

>     **Returns** A dictionary of callback entries or rise a ValueError if an invalid handle is provided.

**process_event**(*namespace*, *data*)
>     Processes an event that has been received either locally or from a plugin.

>     **Parameters**

- **namespace** (*str*) – Namespace the event was fired in.

- **data** – Data associated with the event.

> **Returns** None.

**process_event_callbacks**(*namespace*, *data*)

> Processes a pure event callback.
>
> Locate any callbacks that may be registered for this event, check for filters and if appropriate, dispatch the event for further checking and eventual action.
>
> > **Parameters**
> >
> > - **namespace** (*str*) – Namespace of the event.
> >
> > - **data** – Data associated with the event.
> >
> > **Returns** None.

# 15.8 logging

**class DuplicateFilter**(*logger*, *threshold*, *delay*, *timeout*)

> **filter**(*record*)
>
> > Determine if the specified record is to be logged.
> >
> > Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

**class AppNameFormatter**(*fmt=None*, *datefmt=None*, *style=None*)

> Logger formatter to add 'appname' as an interpolatable field.
>
> **format**(*record*)
>
> > Format the specified record as text.
> >
> > The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

**class LogSubscriptionHandler**(*ad: appdaemon.appdaemon.AppDaemon*, *type*)

> Handle apps that subscribe to logs.
>
> This Handler requires that it's formatter is an instance of AppNameFormatter.
>
> **emit**(*record*)
>
> > Emit a record.
> >
> > If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

# 15.9 main

**class PluginBase**(*ad: appdaemon.appdaemon.AppDaemon*, *name*, *args*)

> Base class for plugins to set up _logging

## 15.10 scheduler

## 15.11 state

## 15.12 stream

## 15.13 thread_async

**class ThreadAsync**(*ad: appdaemon.appdaemon.AppDaemon*)
> Module to translate from the thread world to the async world via queues

## 15.14 threading

## 15.15 utility_loop

Module to handle utility functions within AppDaemon.

**class Utility**(*ad: appdaemon.appdaemon.AppDaemon*)
> Class that includes the utility loop.
>
> Checks for file changes, overdue threads, thread starvation, and schedules regular state refreshes.
>
> **stop**()
> > Called by the AppDaemon object to terminate the loop cleanly
> >
> > > **Returns** None
>
> **loop**()
> > The main utility loop.
> >
> > Loops until stop() is called, checks for file changes, overdue threads, thread starvation, and schedules regular state refreshes.

## 15.16 utils

**class PersistentDict**(*filename*, *safe*, *\*args*, *\*\*kwargs*)
> Dict-like object that uses a Shelf to persist its contents.
>
> **update**($\left[E\right]$, *\*\*F*) → None. Update D from mapping/iterable E and F.
> > If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

**class AttrDict**(*\*args*, *\*\*kwargs*)
> Dictionary subclass whose entries can be accessed by attributes (as well as normally).
>
> **static from_nested_dict**(*data*)
> > Construct nested AttrDicts from nested dictionaries.

**class StateAttrs**(*dict*)

# Upgrading from 3.x

This documentation is for AppDaemon is 4.0.0 or later. If you are upgrading from a 3.x version, there have been some changes to the way AppDaemon is configured, and you will need to edit your config files and make some other changes. The changes are listed below:

Note that not all changes will apply to everyone, some of them are in fairly obscure parts of AppDaemon that few if any people use, however, everyone will have to make some changes, so read carefully.

- `log` section is deprecated in favor of a new and more versatile `logs` section. In AppDaemon 4.x, each log can be configured individually for filename, maximum size, etc. and in addition, it now supports custom formats and additional user logs.

For more detail see the `Log Configuration` section in the Configuration section.

- `api_port` is no longer supported by the `appdaemon` section, it has moved to the new `http` component, and is defined by the port number in the `url` parameter. API Paths to apps have not changed. The App API, Dashboards and new Admin interface all share a single port, configured in the *http* section. For further details, see `Configuring the HTTP Component` in the Configuration section. To turn on support for the App Api, you will need to include an `api` section in AppDaemon.yaml - see the `Configuring the API` section in the Configuration section/

- `latitude`, `longitude`, `elevation` and `timezone` are now mandatory and are specified in the `appdaemon` section of appdaemon.yaml.

# Upgrading from 2.x

This documentation is for AppDaemon is 3.0.0 or later. If you are upgrading from a 2.x version, there have been some changes to the way AppDaemon is configured, and you will need to edit your config files and make some other changes. The changes are listed below:

Note that not all changes will apply to everyone, some of them are in fairly obscure parts of AppDaemon that few if any people use, however, everyone will have to make some changes, so read carefully.

- AppDaemon no longer supports python 3.4

This is a fairly significant step, and the decision to do this was influenced by HASS' pending end of support for Python 3.4. There are many technical reasons why this is a good thing, but the bottom line is that you will need to upgrade your python version to run HASS anyway, so I took this opportunity to follow suit. AppDaemon 3.0 will remain in beta until HASS drops python 3.4 support entirely.

- Apps need to change the import and super class

The naming and placement of the imports needed to change to support the plugin architecture, and to make more sense of the naming in a multiple plugin environment. You will need to edit each of your apps and change the top couple of lines from:

```python
import appdaemon.appapi as appapi

class MyClass(appapi.AppDaemon):
...
```

to:

```python
import hassapi as hass

class MyClass(hass.Hass):
...
```

Note, we are changing both the import name, and the superclass.

- `info_listen_state()` now returns the namespace in addition to the previous parameters

I introduced namespaces as a way of handling multiple plugins at the same time - the docs have more details, but if you are just using a single HASS instance, as everyone has been doing until now, you can safely ignore namespaces.

- The "ha_started" event has been renamed to "plugin_started"

If you use this event, the name has been changed. The plugin started event has a parameter called `name` which gives the name of the plugin that was restarted.

- RSS Feed parameters have been moved to the hadashboard section

When HADashboard is integrated with HASS, the config for HADashboard needs to be all in one place.

e.g.:

```
hadashboard:
  dash_url: http://192.168.1.20:5050
  rss_feeds:
    - feed: http://rss.cnn.com/rss/cnn_topstories.rss
      target: news
  rss_update: 300
```

- Log directives now have their own section

Logging is a function of the underlying execution code, not specifically AppDaemon (for instance, when integrated with HASS, AppDaemon will use HASS logging. For that Reason, the log directives were pulled out into their own section. The section is optional, and if not specified all the previous defaults will apply.

For example:

```
log:
  accessfile: /export/hass/appdaemon_test/logs/access.log
  errorfile: /export/hass/appdaemon_test/logs/error.log
  logfile: /export/hass/appdaemon_test/logs/appdaemon.log
  log_generations: 5
  log_size: 1024
appdaemon:
...
```

- `AppDaemon` section renamed to `appdaemon`, `HADashboard` section renamed to `hadashboard`

This was done mainly for consistency, and because the capitals bugged me ;)

- Plugins (such as the HASS plugin now have their own parameters under the plugin section of the config file

This comes down to a reorganization of the appdaemon.yaml file to reflect the fact that there are now plugins and there may be more than one of them. Rather than having its own section, the HASS plugin is now listed under the `appdaemon` section, although the arguments remain the same. Here is an example:

```
appdaemon:
  api_port: 5001
  api_key: !secret appdaemon_key
  threads: 10
  time_zone: GMT+0BST-1,M3.5.0
  plugins:
    HASS:
      type: hass
      ha_key: !secret home_assistant_key
      ha_url: http://192.168.1.20:8123
      #commtype: SSE
```

- –commtype command line argument has been moved to the appdaemon.cfg file

---

This parameter applies specifically to HASS, so it made no sense to have it as a commandline argument. See above for an example.

- Accessing other Apps arguments is now via the `app_config` attribute, `config` retains just the AppDaemon configuration parameters

Required due to the restructuring of the config files.

- the self.ha_config attribute has been replaced by the `self.get_hass_config()` api call and now supports namespaces.

This reflects the fact that the yaml files have been reconfigured, and that the config info is now owned by the individual plugins.

- The !secret directive has been moved to the top level of appdaemon.yaml

The same argument as the logs - not strictly relevant to AppDaemon, more a concern of the execution environment.

- apps.yaml in the config directory has now been deprecated

One of the new features in 3.0 is that it is now possible to split the apps.yaml into multiple files. You are free to do this in any way you want and place the yaml files with any name, anywhere in the directory hierarchy under the appdir. Apart from flexibility, another reason for this was to prepare the way for later features around configuration tools and automatic app installs. For now, the only necessary step is to move your apps.yaml file from the config directory into the apps directory. If you do not, you will get a warning but everything should still work for now. If you do stick with apps.yaml at in the config directory for now, any other yaml files in the apps directory will be ignored.

- select_value() has been renamed to set_value() to harmonize with HASS

A minor change just to reflect the recent changes to HASS in this area, e.g `input_slider` being renamed to `input_number` and the service name changing.

- It is no longer possible to automatically migrate from the legacy cfg style of config, and support for cfg files has been dropped.

This has been on the cards for a while - if you are still using cfg files, use the latest 2.0 version of appdaemon to migrate to yaml style configuration before you upgrade to 3.0.

- App modules not listed in an apps.yaml file will no longer be loaded. Python modules may still be imported directly if they are in a directory in which other apps reside.

- `cert_path` is deprecated. With the replacement of requests with aiohttp, it is now sufficient to set `cert_verify` to False to use a self signed certificate.

- In apps.yaml, dependencies should now be a proper yaml list rather than a comma separated string

This rewrite introduces some breaking changes as dependencies are now tracked at the app level rather than the module level. This gives a lot more flexibility, and solves a couple of problems. For instance, @ReneTode, the undisputed AppDaemon power user has one App that he is running 60 different instances of. Under the old system, a change to one of those instances parameters in apps.yaml forced all 60 apps to reload - not good :) With the new app level dependencies, just the affected app will reload, along with any other apps that depend on it.

While I was in the code I made another change that I had been wanting to for a while - dependencies used to be a comma separated list, now they are a true yaml list.

So what does that mean for anyone upgrading? Well, if you weren't using dependencies before, then absolutely nothing, all should work the same.

If you were using dependencies, you will need to make some minor changes, to reference apps rather than modules, and to change the format for multiple entries. Here's an example of an old style dependency tree:

```
app1:
  module: module1
  class: class1
app2:
  module: module2
  class: class2
app3:
  module: module3
  class: class3
  dependencies: module1
app4:
  module: module4
  class: class4
  dependencies: module1,module2
```

Under the new system we change the dependencies to apps and change the way the dependencies are listed:

```
app1:
  module: module1
  class: class1
app2:
  module: module2
  class: class2
app3:
  module: module3
  class: class3
  dependencies: app1
app4:
  module: module4
  class: class4
  dependencies:
    - app1
    - app2
```

As you can see, single dependencies can be listed inline, but if you have more than one you must us the YAML list format.

For those of you that are relying on the module based reloading to force reloads of modules that aren't apps, this can be achieved using global module dependencies.

# Change Log

## 18.1  4.0.4

**Features**

- All module dependencies pinned to exact versions for better environmental predictability

- Bump pyyaml to 5.3

- Bump yarl to 1.4.2

- Bump bcrypt to 3.1.7

- Bump jinja2 to 2.10.3

- Bump aiohttp-jinja2 to 1.2.0

- Bump deepdiff from 4.0.9 to 4.2.0

- Bump jinja2 from 2.11.0 to 2.11.1

- Bump deepdiff from 4.2.0 to 4.3.1

- Bump pygments from 2.5.2 to 2.6.1

- Add Azure pipelines for Black and Flake - contributed by Bas Nijholt

- Added service call for `remove_entity`

- Added ability to use `now` in `run_every`. Also seconds can be added by simply using `now+10` for example

- Presence convenience functions now support a `person` flag to use person entities rather than device trackers for presence detection

- `constrain_person` constraints added to support person entities

- Add stream support for SockJS

- Dashboard component now only sends event updates for relevant dashboard entities rather than broadcasting all state_change events

- Admin UI now breaks out App instance and lifetime callback stats separately

- Convert admin and dashboard to get_state from stream

- Increase default work factor for password hashes to 12

- Added *add_entity* api call, alongside *state/add_entity* service call

- Added the ability to remove plugin entities like *HA* when using the *remove_entity* api

- Cleanup sequences when modified. This ensures removed sequences are also removed from the Admin UI and AD

- Added support to use environment variables using the *!env_var* tag, if not wanting to use the *!secrets* tag

- Additional format for time travel start and end times accepted

- Added the ability to specify a callback to hass get_history. This way, large amount of data can be retrieved from the database, without AD cancelling the task

- Added retry_secs parameter to the hass plugin

**Fixes**

- Re-added support for SSL in the http module (should also fix dialogflow)

- Add openssl-dev package to docker image (required for RPI)

- Fixed up socketio support to work with the new stream semantics

- Fixed a bug that allowed multiple copies of an App to run if there was an error in the signature of terminate()

- AppDaemon's REST API no longer needs to be active to use the dashboard or Admin interfaces

- Fix tzdata error in docker build for RPI - contributed by Guy Khmelnitsky

- Fix for *get_tz_offset()* not working in some circumstances - contributed by sillyfrog

- Added some locking to prevent array size change errors

- Fix for registering services created in HA, after it had started

- Added additional logic to wait for full HASS startup

**Breaking Changes**

- Changed `websocket_connected` and `websocket_disconnected` events to `stream_connected` and `stream_disconnected` respectively

- Changed the *get_history* api, as *entity_id* has been removed from the api

None

## 18.2 4.0.3 (2020-02-29)

**Features**

- Pinned astral to v1.10.1

**Fixes**

- Pinned astral to prevent a bug in the latest v2 astral

**Breaking Changes**

None

---

## 18.3  4.0.2 (2020-02-28)

**Features**

None

**Fixes**

- Fixed a critical bug that cause multiple scheduler errors during a leap year - contributed by Chad McCune

**Breaking Changes**

None

## 18.4  4.0.1

**Features**

None

**Fixes**

- Fixed an issue, where when `http` is disabled in `appdaemon.yaml`, AD is unable to start
- Fixed an issue that prevented dashboards from working on older iPads

**Breaking Changes**

None

## 18.5  4.0.0 (2020-01-12)

**Features**

- Added events for when an app is initialized or terminated
- Added *event_fire* service call
- Added *production_mode* service call
- Added *list_services* api call
- Added the ability to fire an event callback only once, using the *oneshot* flag
- Added the ability to use async functions as endpoint callback
- Added the ability for `input_select` to auto-update when the options changes, without need of refreshing the browser page
- Added events for when a websocket client connects and disconnects
- Added the ability for apps to register web routes, thereby utilizing AD's internal web server
- Added static folder *web*, which can used to serve content like images using AD's internal web server
- Added ability for users to define static folders, which can used to serve content like images using AD's internal web server
- Added support for python 3.8

**Fixes**

- Fixed issue where the user could potentially create entities in *admin*, *global* or *appdaemon* namespaces

**Breaking Changes**

None

## 18.6 4.0.0 Beta 2 (2019-10-19)

**Features**

- Added a `timeout` parameter to `listen_state()` and `listen_event()` to delete the callback after a pre-determined interval.

- Added render_template() handling

- global_modules can now be declared in multiple yaml files

- It is now possible to inject arbitrary headers in served http content

- Updated camera widget now supports streams and token refreshing

- Added input_text and input_datetime widgets

- Added the ability to control the number of threadpool workers

- Each time a new service is registered, a `service_registered` event is fired, which can be picked up by apps

- Added support for async apps

- Added authorization to stream as well as command semantics for various functions

- Added sequences

- Added sequence widget

- Added app access to dashboard directory using `self.dashboard_dir`

- List of available dashes is now alphabetically sorted

- Changed namespaces implementation to use shelve instead of JSON enabling non JSON-serializable objects to be stored and also potential performance increases - contributed by Robert Schindler

- MDI updated to version 4.4.95 - contributed by Roeland Van Lembergen

**Fixes**

- Fixed a bug in global_modules that caused a exception

- Fixed icon bug in weather widget - contributed by Roeland Van Lembergen

**Breaking Changes**

- `timeout` is now an official parameter to `listen_state()` and `listen_event()`. If you were using `timeout` in your kwargs section for either you should rename that parameter.

- The camera widget has changed parameters - check the docs for details

- Moved the `log events` from global to `admin` namespace. if `listen_log` is just used for listening to logs, it shouldn't matter

- If you have used persistent namespaces in the previous beta it is necessary to delete all saved namespaces by removing all files in the `namespaces` subdirectory under your appdaemon config directory

# 18.7 4.0.0 Beta1 (2019-08-30)

**Features**

- Apps can now use a simplified version of the import statement e.g. `import hassapi as hass` or `import mqttapi as mqtt`. The existing import method will continue to work.

- Apps can now use multiple plugin APIs with the `get_plugin_api()` function

- Added `ADBase` superclass for apps that want to use the `get_plugin_api()` style of coding

- Scheduler rewritten to be more efficiant and allow for microsecond resolution

- `listen_log()` now sends AppDaemon system messages and has the option to set a log level.

- Bumped aiohttp to v3.4.4

- Added callback locking decorators

- Rearchitected the work Q to allow App pinning and avoid re-entrant and concurrent code if desired

- Implemented multiple worker Ques to avoid Head of Line blocking

- API Calls to control app pinning

- Added the `run_in_thread()` api call - with assistance from Odianosen Ejale

- reworked log listening functions to be more robust and added the ability to have multiple callbacks per app

- Refactored plugin APIs to remove duplication

- Moved `constrain_days` from being Hass only to all app, regardless of plugin used

- Added checking for overdue threads

- Added error checking for callback signatures

- Added app attributes that allows to access AD's `config` and `apps` directories within apps

- Added `parse_datetime()`

- `run_once()`, `run_at()` and `run_daily()` now optionally take `parse_time()` or `parse_datetime()` style arguments for specifying time

- Refactored appdaemon.py for greater readability and easier maintenance

- Expanded on the ability to trigger `listen_state` callbacks immediately using the `immediate` flag, without need of specifying the `new` nor `duration` parameter.

- Allowed to make use of `attribute` when using the `immediate` flag in `listen_state`

- Added initial version of the Admin Interface

- Added User Defined Namespaces

- Rewrote logging to include user defined logs and formats

- Added a unified http component to handle API, ADMIN and DASHBOARD access on a single port

- Added startup conditions to the HASS plugin

- Added duplicate filtering for logs

- Added standalone pidfile functionality

- Added the ability to delete an AD app generated entity from any namespace

- Added the ability to get the history of entities from HASS database

- Added the ability to force a start of the MQTT plugin, even if not connected to broker at startup

- Added the ability to set AD's `production_mode` from within apps

- Added the ability to start, stop, restart and reload apps from either other apps or REST API

- Added the ability to register app services

- Added sensors for different internal state of AD, that can be read by apps

- Added Person widget

- Much reworking of docs

- Added `register_dependency()` for dynamic dependencies in apps

- Added MQTT support for setting TLS version - contributed by Miguel

- Added support for socketio for older tablet devices - inspired by algirdasc and zarya

- Added support for `default` and `copy` parameters in `get_state()` api call - contributed by Robert Schindler

- added a switch to disable the encoding of every log message to ascii - contributed by Ben Lebherz

- Various YAML fixes and refactoring - contributed by Rolf Schäuble

- Allow more natural addition of commandline arguments to Docker and allow spaces - contributed by Christoph Roeder

- Allowed for subscribing to MQTT events using wildcards. e.g. `homeassistant/#` - contributed by Odianosen Ejale

- Allow to specify a MQTT message to be sent when AD shutdowns cleanly e.g. `offline`

- MQTT Retain setting for birth and will messages - contributed by Clifford W. Hansen

- Added Note on long lived tokens for Docker users - contributed by Bob Anderson

- Documentation fixes - contributed by Johann Schmitz

- Documentation fixes - contributed by Brendon Baumgartner

- Documentation fixes - contributed by Quentin Favrie

- Documentation fixes, updating and cleaning - contributed by Humberto Rodríguez A.

- Added the ability to set title 2 as friendly name in widgets - contributed by Radim

- Added the ability to listen to `state_change` events, without using listen_state() - contributed by Thomas Delaet

- APIAI updated to dialog flow - contributed by engrbm87

**Fixes**

- Fixes to listen_state() oneshot function

- Fixes to listen_state() oneshot function when duration is used

- Fixes to listen_state() function when it fires even when new and old states are same

- Fixed an issue causing incorrect busy thread counts when app callbacks had exceptions

- Fixed an issue of when MQTT Plugin not connected to broker, and it holds up AD startup

- Fix to Forecast min/max in weather widget - contributed by adipose

- Fix climate widget docs - contributed by Rene Tode

- Fix to harmonize `units` vs `unit` - contributed by Rene Tode

- Added missing import in sound.py example - contributed by cclaus

- Fix for run_once() - contributed by engrbm87

- Fix for onclick not working on IE11 - contributed by jgrieger1

- Fixed issue of AppDaemon loading all `.yaml` files, even those starting with a `.` which are hidden or binary files. Contributed by fhirschmann

- Fix for error generated when a none existent schedule timer is passed to `info_timer`

- Fix for `log_type` flag in `listen_log` callback

- Relative paths for appdaemon's config directory now work correctly

- Fix to Dialogflow after format changes

- MQTT fix to subscribing using wildcards - contributed by Daniel Lashua

**Breaking Changes**

- appapi.py has been renamed to adbase.py, and the contained superclass ha been renamed from AppDaemon to ADBase. This should only be a breaking change if you were using unpublished interfaces!

- Time travel semantics have changed to support faster scheduling.

- `plugin_started` and `plugin_stopped` now go to the appropriate namespace for the plugin and are no longer global

- Apps are no longer concurrent or re-entrant by default. This is most likely a good thing.

- Changed the signature of `listen_log()` callbacks

- `cancel_listen_log()` now requires a handle supplied by the initial `listen_log()`

- Removed Daemonize support - please use sysctl instead

- `set_app_state()` is deprecated - use `set_state()` instead and it should do the right thing

- `dash_compile_on_start` now defaults to true

- The `log` section of appdaemon.yaml has been deprecated and must be replaced by the new `logs` section which has a different format to allow for user defined logs and greater flexibility in formatting etc.

- API no longer has a separate port, all access is configured via the new unified http component

- API has its own top level configuration section

- Some dashboard parameters moved to the `HTTP` section and renamed

- `dash_compile_on_start` renamed to `compile_on_start`

- `dash_force_compile` renamed to `force_compile`

- Due to the new `log` parameter to allow apps to use user defined logs, any previous parameters named `log` should be renamed

- Due to a fix for `info_timer`, this function can now return `None` if the timer handle is invalid

- As a result of a change in the way AD auto generates MQTT client status topic, if not defined previously the new topic needs to be used

- In the appdaemon configuration section, `latitude`, `longitude`, `elevation` and `timezone` are now mandatory

- MQTT client status api change from `clientConnected` to `is_client_connected`

## 18.8  3.0.4 (2019-04-04)

**Fixes**

- Use yaml.Safeloader to work around known security issue with PyYaml - contributed by mvn23
- Unpinned PyYaml

## 18.9  3.0.3 (2019-04-02)

**Fixes**

- Pinned PyYaml to 3.13 to avoid a known issue

## 18.10  3.0.2 (2018-10-31)

**Features**

- added `set_textvalue()` api call.
- added `app_init_delay` to delay App Initialization
- Added ability to register apps to receive log entries
- Added instructions for running a dev build
- Added support for Long Lived Access Tokens
- Updated MDI Icons to 3.0.39
- Updated Font Awesome Icons to 5.4.2
- Added MQTT Plugin - contributed by Tod Schmidt
- Many MQTT Plugin enhancements - contributed by Odianosen Ejale
- Added `entitypicture` widget - contributed by hwmland
- Docker start script will now check recursively for additional requirements and install them - contributed by Kevin Eifinger
- Added ability to set units explicitly in widgets - contributed by Rene Tode
- Added –upgrade to pip3 call for recursive requirements.txt scanning - contributed by Robert Schindler
- Added the ability to pass stringified JSON parameters to service calls - contributed by Clyra

**Fixes**

- Fixed incorrect service call in `set_value()`
- Enforce domain name in rss feed target to avoid issues with other functions
- Previously deleted modules will now be correctly reloaded to reflect changes
- Fixed a bug in `get_scheduler_entries()`
- Prevent periodic refresh of HASS state from overwriting App created entities - contributed by Odianosen Ejale
- Fix to honor cert_path - contributed by Myles Eftos
- Run AD in docker as PID 1 - contributed by Rolf Schäuble

- Fix encoding error in log messages - contributed by Markus Meissner
- Fix a bug in `get_plugin_meta()` - contributed by Odianosen Ejale
- Various Doc corrections and additions - contributed by Odianosen Ejale
- Various fixes in the Docker docs - contributed by Simon van der Veldt
- Namespace fixes - contributed by Odianosen Ejale
- More namespace fixes - contributed by Odianosen Ejale
- Fixes of the namespaces fixes ;) - contributed by Brian Redbeard
- Fix typo in sample systemd config - contributed by Evgeni Kunev
- Fix to cert path config - contributed by nevalain

**Breaking Changes**

- RSS target names must now consist of a domain as well as the target name, e.g. `rss.cnn_news`
- SSE Support has been removed
- Use of ha_key for authentication is deprecated and will be removed at some point. For now it will still work
- Many Font Awesome Icon names have changed - any custom icons you have on dashboards will need to be changed to suit - see docs for more detail.

While working through the upgrade it is strongly advised that you clear your browser cache and force the recompilation of all of your dashboards to flush out references to old icons. This can be done by manually removing the `compiled` subdirectory in `conf_dir`, specifying `recompile=1` in the arguments to the dashboard, or setting the hadashboard option `dash_compile_on_start` to `1`.

## 18.11 3.0.1 (2018-04-18)

**Features**

- Added Production Mode to disable checking of App config or code changes
- RSS Feed can now optionally show a description for each story
- Disabling of zooming and double tap zooming on iOs devices is now optional via the `scaling` dashboard argument
- Exiting from the commandline with ctrl-c will now cleanly terminate apps
- Sending SIGTERM to an appdaemon process will cause a clean shutdown, including orderly termination of all apps in dependency order
- Added extra checking for HASS Initialization to prevent a race condition in which metadata could not be read
- Weather widget adds the ability to change sensors, more dynamic units, forecast option, icon options, option to show Rain/Snow depending on precip_type sensor (and change icons), wind icon rotates according to wind bearing - contributed by Marcin Domański

**Fixes**

- Fixed a problem in the Docker initialization script
- Fixed an parameter collision for events with a parameter `name` in `listen_event()`
- Grammar corrections to docs, and a fix to the stop code - contributed by Matthias Urlichs

**Breaking Changes**

- iOS Scaling and tap zooming is no longer disabled by default

## 18.12 3.0.0 (2018-03-18)

**Features**

- API 200 responses are now logged to the access file
- Add meta tags to prevent double tap zoom on iOS

**Fixes**

- Re-added set_app_state() to the API

**Breaking Changes**

None

## 18.13 3.0.0b5 (2018-03-05)

**Features**

- Added additional error checking for badly formed RSS feeds

**Fixes**

- Fixed a bug that broke binary_sensor widget.
- Fixed a bug that broke retries when connecting to Home Assistant
- Fixed a bug that could cause lockups during app initialization
- Fixed a bug for Docker that prevented the initial config from working correctly - contributed by mradziwo
- Grammar corrections to docs, and a fix to the stop code - contributed by Matthias Urlichs

**Breaking Changes**

None

## 18.14 3.0.0b4 (2018-03-03)

**Features**

- Single App dependencies can now be specified on the dependency line itself and don't have to be a list of size 1
- Added `get_ad_version()`, and `ad_version` to the config dictionary
- Added filters for Apps
- Added global module dependency tracking
- Added plugin reload app control
- Added icon widget

**Fixes**

- Apps now correctly reload when HASS comes back up after a restart
- `get_error()` now properly returns the error log logger object

- `get_hass_config()` is now correctly named

- `app_args` now correctly returns args for all apps

- `get_state()` now returns fields from the attributes dictionary in preference to the top level dictionary if there is a clash. In particular, this now means it is easier to iterate through group members

- Fixed a bug preventing an objects `terminate()` from being called when deleted from apps.yaml

- Fixed a bug in which object info was not being cleaned out at object termination

- Fixed an issue preventing dashboard updates on python 3.6

**Breaking Changes**

None

## 18.15  3.0.0b3 (2018-02-11)

**Features**

- Added `javascript` widget

- Upgraded MDI Icons to 2.1.19

- Add separate log for diagnostic info

- Per-widget type global parameters

- App level dependencies

- `listen_state()` now returns the handle to the callback

- added `oneshot` option to `listen_state()`

- Add step parameter to climate widget - contributed by Adrian Popa

- Add internationalization options to clock widget - contributed by Adrian Popa

- Doc improvements - contributed by Marco

**Fixes**

- Fixed image path for android devices

- Fix a bug with the time parameter for images

- Fixed `disable_apps`

- Fixed a bug in `get_state()` with `attributes=all` returning just the attributes dictionary instead of the entire entity.

**Breaking Changes**

- In apps.yaml, dependencies should now be a proper yaml list rather than a comma separated string

- Dependencies now refer to individual apps rather than modules

## 18.16  3.0.0b2 (2018-01-27)

**Features**

- Make int args in appdaemon.yaml a little more robust

- Improve handling for missing app files

- Module loading enhancements

- Moved from requests to aiohttp client for better async behavior

- Added thread monitoring for worker threads

- Give more informative error message if AppDaemon can't locate a valid config dir

**Fixes**

- Fixed a bug that could cause multiple apps.yaml changes or additions to be ignored

- Fixed a bug causing listen_state() callbacks with `duration` set to fire immediately

- Pinned yarl library to fix an issue with Docker build

- Fixed a couple of potential event loop hold ups

- Fixed a bug in password security for HADashboard service and state calls

- Changes to apps.yaml now also force a reload of dependent modules

- `exclude_dirs` now applies to yaml files as well as python files

- Fixed broken icon on HADashboard logon screen

- Fixed a bug preventing the media title from showing in the media player

**Breaking Changes**

- App modules not listed in an apps.yaml file will no longer be loaded. Python modules may still be imported directly if they are in a directory in which other apps reside.

- `cert_path` is deprecated. With the replacement of requests with aiohttp, it is now sufficient to set `cert_verify` to False to use a self signed certificate.

- Initial dashboard loads may be slower on less powerful hardware when using password authentication. Updating after the initial load is unaffected.

## 18.17  3.0.0b1 (2018-01-12)

**Features**

- Refactored pluggable architecture

- Support for multiple HASS instances

- Custom constraints

- Namespaces

- Path of Secret file can now be specified

- apps.yaml can now be split across multiple files and directories

- Apps can now establish loading priorities to influence their loading order

- IFRAME Refreshes should now be more reliable

- Added calls to access the underlying logger objects for the main and error logs

- Add the ability to ignore specific subdirectories under appdir

- Added error handling for apps that can't be read or have broken links

- Added london Underground Widget - contributed by mmmmmmtasty

- Added ability to display sensor attributes - contributed by mmmmmmtasty

- Added Weather Summary Widget - contributed by mmmmmmtasty

- Added Sticky navigation - contributed by Lars Englund

- Added Input Select widget - contributed by Rene Tode

- Redesigned Input Number widget (old is still available as `input_slider`) - contributed by Rene Tode

- Added Radial widget - contributed by Rene Tode

- Added Temperature widget - contributed by Rene Tode

- Added container style to sensor widget - contributed by Rene Tode

**Fixes**

- Fixed an issue with the compiled directory not being created early enough

**Breaking Changes**

- Apps need to change the import and super class

- `info_listen_state()` now returns the namespace in addition to the previous parameters

- AppDaemon no longer supports python 3.4

- –commtype command line argument has been moved to the appdaemon.cfg file

- The "ha_started" event has been renamed to "plugin_started"

- RSS Feed parameters have been moved to the hadashboard section

- Log directives now have their own section

- *AppDaemon* section renamed to *appdaemon*, *HADashboard* section renamed to *hadashboard*

- Accessing other Apps arguments is now via the `app_config` attribute, `config` retains just the AppDaemon configuration parameters

- Plugins (such as the HASS plugin now have their own parameters under the plugin section of the config file

- The !secret directive has been moved to the top level of appdaemon.yaml

- the self.ha_config attribute has been replaced by the `self.get_hass_config()` api call and now supports namespaces.

- apps.yaml in the config directory has now been deprecated

- select_value() has been renamed to set_value() to harmonize with HASS

- It is no longer possible to automatically migrate from the legacy cfg style of config, and support for cfg files has been dropped.

## 18.18 2.1.12 (2017-11-07)

**Features**

None

**Fixes**

- Fixed passwords causing 500 error on HADashboard - contributed by wchan.ranelagh

**Breaking Changes**

None

## 18.19 2.1.11 (2017-10-25)

**Features**

None

**Fixes**

- Fixed an issue with `run_at_sunset()` firing multiple times

**Breaking Changes**

None

## 18.20 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

## 18.21 2.1.9 (2017-09-08)

**Features**

None

**Fixes**

- broken *disable_apps* temporary workaround

**Breaking Changes**

None

## 18.22 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 focused 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

## 18.23 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()*

## 18.24 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`

## 18.25  2.1.5 (2017-08-10)

**Features**

None

**Fixes**

None

**Breaking Changes**

- `get_alexa_slot_value()` now requires a keyword argument for slotname

## 18.26  2.1.4 (2017-08-10)

**Features**

None

**Fixes**

- .cfg file fixes

**Breaking Changes**

None

## 18.27  2.1.3 (2017-08-10)

**Features**

- Restructure docs for readthedocs.io

None

**Fixes**

None

**Breaking Changes**

None

## 18.28  2.1.2 (2017-08-08)

**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

## 18.29  2.1.0 (2017-08-08)

**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

## 18.30  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

## 18.31 2.0.7 (2017-07-20)

**Features**

None

**Fixes**

- Fixed a bug in label and text_sensor widgets

**Breaking Changes**

None

## 18.32 2.0.6 (2017-07-20)

**Features**

None

**Fixes**

- Fix a bug causing an apps `terminate()` to not be called

**Breaking Changes**

None

## 18.33 2.0.5 (2017-07-16)

**Features**

None

**Fixes**

- Change `convert_utc()` to use iso8601 library

**Breaking Changes**

None

## 18.34 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

## 18.35  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

## 18.36  2.0.2 (2017-07-08)

**Features**

- Move docker image to python 3.6

**Fixes**

None

**Breaking Changes**

None

## 18.37  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

## 18.38 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 arbitrarily complex nested data structures in App parameters
- Added ability to convert from old cfg file to YAML
- AppDaemon Apps can now publish arbitrary 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

## 18.39 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**

-

# 18.40 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

# 18.41 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`

## 18.42 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`

## 18.43 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

## 18.44  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

## 18.45  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

## 18.46  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

## 18.47 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

## 18.48 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 necessary 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

## 18.49 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

## 18.50  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

## 18.51  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

## 18.52  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

## 18.53 1.3.3 (2016-09-16)

**Features**

- Add ability to specify a cert directory 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

## 18.54 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

## 18.55 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

## 18.56  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`

## 18.57  1.2.2 (2016-08-31)

**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

## 18.58  1.2.1 (2016-08-26)

**Features**

- Add support for windows

**Fixes**

None

**Breaking Changes**

None

## 18.59 1.2.0 (2016-08-24)

**Features**

- Add support for recursive directories - suggested by jbardi

**Fixes**

None

**Breaking Changes**

None

## 18.60 1.1.1 (2016-08-23)

**Fixes**

- Fix init scripts

## 18.61 1.1.0 (2016-08-21)

**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 specified 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

## 18.62 1.0.0 (2016-08-09)

**Initial Release**

Index

## 19.1 A

## 19.2 B

## 19.3 C

- get_history() [Hass API]
- get_main_log() [AppDaemon API]
- get_namespace() [AppDaemon API]
- get_now() [AppDaemon API]
- get_now_ts() [AppDaemon API]
- get_pin_thread() [AppDaemon API]
- get_plugin_config() [AppDaemon API]
- get_scheduler_entries() [AppDaemon API]
- get_state() [AppDaemon API]
- get_thread_info() [AppDaemon API]
- get_timezone() [AppDaemon API]
- get_tracker_details() [Hass API]
- get_tracker_state() [Hass API]
- get_trackers() [Hass API]
- get_tz_offset() [AppDaemon API]
- get_user_log() [AppDaemon API]
- group [Widget]

## 19.8 H

## 19.9 I

- icon [Widget]
- iframe [Widget]
- info_listen_event() [AppDaemon API]
- info_listen_state() [AppDaemon API]
- info_timer() [AppDaemon API]
- input_boolean [Constraints]
- input_boolean [Widget]
- input_number [Widget]
- input_select [Constraints]
- input_select [Widget]
- input_slider [Widget]
- is_client_connected() [Mqtt API]

## 19.10 J

- javascript [Widget]

## 19.11 K

## 19.12 L

- label [Widget]
- light [Widget]
- list_namespaces() [AppDaemon API]
- list_services() [AppDaemon API]
- listen_event() [AppDaemon API]
- listen_event() [Mqtt API]
- listen_log() [AppDaemon API]
- listen_state() [AppDaemon API]
- lock [Widget]
- log() [AppDaemon API]
- london_underground [Widget]

## 19.13 M

- media_player [Widget]
- mode [Widget]
- mqtt_publish() [Mqtt API]
- mqtt_subscribe() [Mqtt API]
- mqtt_unsubscribe() [Mqtt API]

## 19.14 N

- navigate [Widget]
- noone_home() [Hass API]
- notify() [Hass API]
- now_is_between() [AppDaemon API]

## 19.15 O

## 19.16 P

- parse_datetime() [AppDaemon API]
- parse_time() [AppDaemon API]
- parse_utc_string() [AppDaemon API]
- person [Widget]
- presence [Constraints]

## 19.17 Q

## 19.18 R

- radial [Widget]
- register_endpoint() [AppDaemon API]
- register_route() [AppDaemon API]
- register_service() [AppDaemon API]
- reload [Widget]
- reload_apps() [AppDaemon API]
- remove_entity() [AppDaemon API]
- restart_app() [AppDaemon API]
- rss [Widget]
- run_at() [AppDaemon API]
- run_at_sunrise() [AppDaemon API]
- run_at_sunset() [AppDaemon API]
- run_daily() [AppDaemon API]
- run_every() [AppDaemon API]
- run_hourly() [AppDaemon API]
- run_in() [AppDaemon API]
- run_in_thread() [AppDaemon API]
- run_minutely() [AppDaemon API]
- run_once() [AppDaemon API]

## 19.19 S

- save_namespace() [AppDaemon API]
- scene [Widget]
- script [Widget]
- select_option() [Hass API]
- sensor [Widget]
- set_app_pin() [AppDaemon API]
- set_error_level() [AppDaemon API]
- set_log_level() [AppDaemon API]
- set_namespace() [AppDaemon API]
- set_pin_thread() [AppDaemon API]
- set_production_mode() [AppDaemon API]
- set_state() [AppDaemon API]
- set_textvalue() [Hass API]
- set_value() [Hass API]
- split_device_list() [AppDaemon API]
- split_entity() [AppDaemon API]
- start_app() [AppDaemon API]
- stop_app() [AppDaemon API]
- sun_down() [AppDaemon API]
- sun_up() [AppDaemon API]
- sunrise() [AppDaemon API]
- sunset() [AppDaemon API]
- switch [Widget]

## 19.20 T

- temperature [Widget]
- time [Constraints]
- time() [AppDaemon API]
- toggle() [Hass API]
- turn_off() [Hass API]
- turn_on() [Hass API]

## 19.21 U

- unregister_endpoint() [AppDaemon API]
- unregister_route() [AppDaemon API]

## 19.22 V

## 19.23 W

- weather [Widget]
- weather_summary [Widget]

## 19.24 X

## 19.25 Y

## 19.26 Z

# Python Module Index

## a

# Index