gardnr Documentation

Release 0.5.3

Jason Biegel

Dec 01, 2019

Contents:

GARDNR is a bootstrapper for DIY IoT projects with a focus on horticulture. What that means is GARDNR makes it easy to setup customized fully-functional monitoring and automation solutions. All of the mundane features of creating a system come out-of-the-box. This allows you to focus on writing the code that interfaces with hardware.

CHAPTER 1

Tutorials

Here is the best place to start learning how to get GARDNR up and running. The tutorial sections build off of each other so it will be easier to do them in order.

1.1 Part 1: Logging metric data with Drivers

GARDNR requires Python 3.5 or higher.

Be sure to famliarize yourself with Object-oriented programming and Python classes before using GARDNR.

First, install GARDNR using pip:

```
$ pip install gardnr
```

The basic features of GARDNR are logging metrics to the database and exporting metric logs. To start logging a metric, you first must add the metric to the database, like so:

```
$ gardnr add metric air temperature hello-world
```

Next, a sensor driver can be added which can create logs for the metric *hello-world*. The sensor driver code must be implemented before it can be added to the database. GARDNR can generate empty templates of driver classes to be able to write them faster.

```
$ gardnr new sensor hello_world_sensor.py
```

There should now be a file called *hello_world_sensor.py* in your current directory. Open this file in your preferred code editor, it should contain:

```
from gardnr import drivers, logger, metrics


class Sensor(drivers.Sensor):
    """
```

(continues on next page)

(continued from previous page)

```
    Add code to interface with physical or virtual sensors here.
    """

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def read(self):
        """
        Example log:

        metrics.create_metric_log('my-metric', 42)


        Example image log:

        metrics.create_image_log(
            self.metric_name,
            image_bytes,
            extension=self.image_file_extension
        )
        """

        # remove the next line and add code
        pass
```

At the end of the file, remove the last two lines and insert:

```
metrics.create_metric_log('hello-world', 20)
```

Be sure to indent the line above by eight spaces so it is properly nested under the *read* method. Your *hello_world_sensor.py* file should now look like:

```
from gardnr import drivers, logger, metrics


class Sensor(drivers.Sensor):
    """
    Add code to interface with physical or virtual sensors here.
    """

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def read(self):
        """
        Example log:
```

(continues on next page)

(continued from previous page)

```
        metrics.create_metric_log('my-metric', 42)


        Example image log:

        metrics.create_image_log(
            self.metric_name,
            image_bytes,
            extension=self.image_file_extension
        )
        """

        metrics.create_metric_log('hello-world', 20)
```

Next, the sensor driver module must be added to GARDNR's system. To do this, run the following command:

```
$ gardnr add driver hello-world-sensor hello_world_sensor:Sensor
```

The sensor driver module you just added to GARDNR can now be executed using the following command:

```
$ gardnr read
```

What running the command above does is create a log for our *hello-world* metric. Now we can add an exporter driver to GARDNR. Exporters allow logs to be sent to external locations.. In this case, the log will simply be printed to the console for demostration purposes. First, start with an empty exporter template:

```
$ gardnr new exporter hello_world_exporter.py
```

Next, open *hello_world_exporter.py* in your preferred code editor, it should contain:

```
from gardnr import constants, drivers, logger


class Exporter(drivers.Exporter):
    """
    Uncomment these to filter the types of metrics are logged.
    Either whitelist or blacklist must be used, not both.
    """
    # whitelist = [constants.IMAGE]
    # blacklist = [constants.IMAGE]

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def export(self, logs):
        """
        Output the list of logs to an external destination.
        The log object has the following fields available.

        Log:
```

(continues on next page)

(continued from previous page)

```
            id: str
            timestamp: datetime
            longitude: float
            latitude: float
            elevation: float
            value: blob
            metric:
                topic: str
                type: str
                manual: bool
        """
        for log in logs:
            # remove the next line and add code
            pass
```

At the end of the file, remove the last two lines and insert:

```
print(log.value)
```

Be sure to indent the line above by 12 spaces so it is properly nested under the *for* loop inside the *export* method. Your *hello_world_sensor.py* file should now look like:

```
from gardnr import constants, drivers, logger


class Exporter(drivers.Exporter):
    """
    Uncomment these to filter the types of metrics are logged.
    Either whitelist or blacklist must be used, not both.
    """
    # whitelist = [constants.IMAGE]
    # blacklist = [constants.IMAGE]

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def export(self, logs):
        """
        Output the list of logs to an external destination.
        The log object has the following fields available.

        Log:
            id: str
            timestamp: datetime
            longitude: float
            latitude: float
            elevation: float
            value: blob
            metric:
                topic: str
                type: str
```

(continues on next page)

(continued from previous page)

```
            manual: bool
        """
        for log in logs:
            print(log.value)
```

Next, the exporter driver module must be added to GARDNR's system. To do this, run the following command:

```
$ gardnr add driver hello-world-exporter hello_world_exporter:Exporter
```

The exporter driver module you just added to GARDNR can now be executed using the following command:

```
$ gardnr write
```

You should now see *20* displayed in the console. Note, that if you were to run the above command again, nothing would be displayed. This is because metric logs are only exported once per exporter in the system.

1.2 Part 2: Manual Log Entry

In *Part 1: Logging metric data with Drivers* you created a driver which wrote logs for an air temperature metric, *hello-world*. For some metrics, you may not have the sensor hardware available to read metric data so there will be no driver code to write to log metric data. Or, sensor hardware may fail and your driver code is unable to log metric data. In these cases, you will want to manually log your metric data. To set a metric to manual mode, run:

```
$ gardnr manual on hello-world
```

Manual metrics will be displayed on the Log Entry System. The Log Entry System is part of the GARDNR web server, to start it run:

```
$ gardnr-server -b 0.0.0.0:5000
```

Next, open http://localhost:5000 in a browser. You should see a text field to enter a value for the *hello-world* metric. After you entered a number into the text field, hit the Submit buttom to log the value.

1.3 Part 3: Scheduling

Sensor and exporter drivers can be executed manually using the *read* and *write* commands respectively:

```
$ gardnr read  # executes sensor drivers
$ gardnr write # executes exporter drivers
```

This can be tedious and inconvenient to manually run commands in a shell to execute driver code. Schedules can be used to automatically execute drivers. First a schedule must be created. Schedules can be added to GARDNR using CRON syntax.

```
$ gardnr add schedule every-five-minutes \*/5 \* \* \* \*
You entered: Every 5 minutes
Does this look correct? ([y]/n)
```

Note that the `\*` is to escape the astericks so it is not evaluated by the shell as a wildcard. The command above will add a schedule named *every-five-minutes* after you confirm by typing `y` and then `Enter`. Next, schedule a driver. We will schedule the *hello-world-sensor* we created in *Part 1: Logging metric data with Drivers*:

```
$ gardnr schedule add hello-world-sensor every-five-minutes
```

To execute scheduled drivers, enter the following command which will run indefinitely:

```
$ gardnr-automata
```

1.4 Part 4: Power Drivers

Power Drivers are used to control devices with binary states, either on or off. To create a new power driver from an empty template, run:

```
$ gardnr new power hello_world_power.py
```

There should now be a file called *hello_world_power.py* in your current directory. Open this file in your preferred code editor, it should contain:

```
from gardnr import drivers, logger


class Power(drivers.Power):
    """
    Add code to interface with physical powered devices.
    """

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def on(self):

        """
        Communicate with a powered device

        Example:

        import subprocess
        command = 'gpio mode 0 out; gpio write 0 1'
        process = subprocess.Popen(command.split(), stdout=subprocess.PIPE)
        output, error = process.communicate()
        """

        # remove the next line and add code
        pass

    def off(self):
        pass
```

In the `on` method, remove the last two lines and insert:

```
print('Device turning on')
```

In the `off` method, remove the last line and insert:

```
print('Device turning off')
```

Your *hello_world_power.py* file should now look like:

```
from gardnr import drivers, logger


class Power(drivers.Power):
    """
    Add code to interface with physical powered devices.
    """

    def setup(self):
        """
        Add configuration here
        """

        # remove the next line and add code
        pass

    def on(self):

        """
        Communicate with a powered device

        Example:

        import subprocess
        command = 'gpio mode 0 out; gpio write 0 1'
        process = subprocess.Popen(command.split(), stdout=subprocess.PIPE)
        output, error = process.communicate()
        """

        print('Device turning on')

    def off(self):
        print('Device turning off')
```

Next, add the driver to GARDNR by running:

```
$ gardnr add driver hello-world-power power_driver:Power
```

To run the `on` method, run the following command:

```
$ gardnr power on hello-world-power
```

You should now see *Device turning on* displayed in the console. To run the `off` method, run the following command:

```
$ gardnr power off hello-world-power
```

You should now see *Device turning off* displayed in the console.

Like sensor and exporter drivers, power drivers can also be scheduled, which is described in *Part 3: Scheduling*. However, adding schedules for power drivers requires specifying the state as well. To put turning on a power driver on a schedule, run:

```
$ gardnr schedule add hello-world-power every-five-minutes on
```

1.5 Part 5: Grows

Grow models are used to group metric logs into a collection, ideally for the duration of a single crop cycle. Grows consist of a start and end time. To start a grow, run:

```
$ gardnr grow start
```

This will create an active grow until the grow is ended. Having an active grow also enables the use of triggers, which is described in *Part 6: Triggers and Grow Recipes*

1.6 Part 6: Triggers and Grow Recipes

Triggers are a way to automatically change the state of a power driver based on the rules defined in a grow recipe. In order for triggers to be enabled, there must be an active grow, explained in *Part 5: Grows*, as well as a grow recipe file configured. Grow recipes are stored in an XML files and adhere to the Grow Recipe Schema.

To start, create a very simple grow recipe in a text editor:

```
<?xml version="1.0" encoding="UTF-8"?>

<recipe>

  <default>
    <air>
      <temperature min="16" max="18" />
    </air>
  </default>

</recipe>
```

Save this file as *hello-world-recipe.xml*. Next, configure GARDNR to use the recipe you just created by adding it to the settings. Open a new blank file in a text editor and add:

```
GROW_RECIPE = 'hello-world-recipe.xml'
```

Save the file as *settings_local.py*.

Next, add a trigger on the *hello-world* metric we created in *Part 1: Logging metric data with Drivers* when it reaches the max temperature we specified in the grow recipe, it turns on the *hello-world-power* driver specified in *Part 4: Power Drivers*. To add the trigger, run:

```
$ gardnr add trigger hello-world max hello-world-power on
```

Triggers are checked on a special schedule when `gardnr-automata` is run. To test, make sure `gardnr-automata` is running and run a `gardnr read` command to log a metric value of 20 for the *hello-world* metric. Within five minutes, you should see 'Device turning on' appear in the console running `gardnr-automata` because a temperature of 20 exceeds the max threshold of 18 for the air temperature metric *hello-world*, specified in the grow recipe.

Congratulations! You now know all of the core features GARDNR has to offer. Now you can start writing drivers which interface with actual hardware devices and data stores. Some out-of-the-box drivers are available on GitHub with instructions on setting up the hardware.

CHAPTER 2

Guides

Here is an assortment of guides for more advanced features of GARDNR not covered in the tutorial

2.1 Driver Configuration

Driver configuraiton is used to make `Driver` classes more re-usable. Instead of hardcoding attributes into a class, you can have the attribute loaded at run-time into the driver instance. For example, imagine you have a sensor driver class:

```
from gardnr import drivers


class Sensor(drivers.Sensor):

    def read(self):
        print('hello world')
```

Instead of using the string literal, `'hello world'`, an attributes can be used instead:

```
from gardnr import drivers


class Sensor(drivers.Sensor):

    def read(self):
        print(self.message)
```

Now, to set the the `message` attribute, we pass in a value while adding the driver class to GARDNR

```
$ gardnr add driver sensor attribute_sensor:Sensor -c message="hello world"
```

2.1.1 Multiple configuration attributes

Multiple configuration attributes can be passed in while adding a driver to GARDNR

```
$ gardnr add driver sensor attribute_sensor:Sensor -c attr1=55 attr2="foo"
```

2.2 MQTT

GARDNR comes with a built-in MQTT subscriber for logging metrics. In order to use the subscriber, it must be connected to a broker, such as Mosquitto.

Use the metric name as the topic and the log value as the message.

To start the MQTT subscriber, run `gardnr-mqtt`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search