neat Documentation

Author

Aug 16, 2021

Contents:

Appalachian State University's networked energy appliance translator

CHAPTER 1

Introductions

Authors Sierra Milosh, Stephen Bunn, Nathan Davis, James Ward

1.1 Who We Are

We are two senior and two graduate level students at Appalachian State University under the SSTEM NSF funded scholarship. Each semester, we get to choose a research project to work on with other students from the SSTEM program. This semester, we chose to help create a dashboard for a student-run, student-funded group on campus called the Renewable Energy Initiaitve.

1.2 Why We're Doing This Project

The Renewable Energy Initiative at Appalachian State University gets a $5, self-imposed fee from each student per semester to put towards renewable energy systems on campus. Recently, the REI passed a referendum that required each new project proposal include in the plan and budget a data monitoring system. Over the past few years, the REI has been putting a lot more time, effort, and money into monitoring all of the renewable energy systems on campus, including PV (photovoltaic or solar) systems, solar thermal systems, and a wind turbine (previously the largest turbine in North Carolina, to boot). We want to be able to track data on each system for several reasons:

1. **So that we can see when a system isn't running properly and begin to address the problem.** The monitoring systems track anywhere between 10 and 50 points of data per device, so the ability to see both live and historical data allows for a more comprehensive analysis of what might be going wrong within the system.
2. **So that we can use the data to tell a story to people.** We want our students to be involved with sustainability on campus, so using real numbers to tell a story about our renewable energy systems on campus (that their student funds have paid for) is on the top of our priority list. We would like to be able to talk about how much energy each system is producing, and how that energy production ranks among the other renewable energy systems on campus.
3. **So that we have easily accessible data to be used for reports.** Appalachian's commitment to sustainability is ranked among hundreds of other universities in the country through massive reports like the STARS report

and the Greenhouse Gas Inventory. We would like to be able to go in and easily access data to fill in these comprehensive reports.

But primarily, the goal of the REI is to get students involved with renewable energy that they are funding on campus. We want to have clean, historical, and live data to bring it down a level and talk to students about what is happening on their campus.

Currently, the Renewable Energy Initiative is pouring in $10,000 from our $150,000 budget to (unnamed company) to create a dashboard that displays the data from our systems on campus in understandable graphs. The REI is unsatisfied with the current company, as:

- Data is not technically live (it uploads via FTP every 15 minutes).
- The graphs are not very customizable – The user cannot go in and add whichever features they please to any graph – They instead have to set up a call with (unnamed company) to try to get those features added, and (unnamed company) does not always know how to add the desired features.
- The REI has to create virtual meters because the (unnamed company) does not support certain devices – The REI had to consolidate all of the information into one big table with all of the desired devices and units.

CHAPTER 2

Getting Started

2.1 Developers

The following subsections detail what is required for various tasks during development.

2.1.1 Licensing

The `neat` framework is licensed under the GNU GPLv3 license. This is a strong copyleft license which basically means that permissions are conditioned on making available **complete** source code of licensed works and modifications (including larger works). This license was chosen with upmost care as we feel that the potential of this project may encourage future use of renewable energy appliances in conjunction with this system. We found our decision on the basis that any form of software built to aid the future of renewable energy adoption should be free and open to the public for consumption.

2.1.2 Versioning

The `neat` framework strictly follows Semantic Versioning 2.0.0 as proposed by Tom Preston-Werner. The in-house development period is to follow the 0.x.x standard until the initial release of a full scale product (at which time will change to its first major release).

2.1.3 Coding Conventions

NEAT source follows the PEP8 - Style Guide for Python Code the more recently named pycodestyle. The only exception to this style guide is the rule on line length. This rule has been omitted simply because of its occasional annoyance. Code written in in this project should still try to adhere to the 79 character limit while documentation should stay under the 72 character limit.

You can disable the checking of line length by passing the error code `E501` as a value into the ignore list of pep8. For example `pep8 --ignore=E501 ./`. We highly recommend you install a linter plugin for you editor that follows the pycodestyle (pep8) format.

2.1.4 Documentation Conventions

In-code documentation utilizes Python's docstrings but does not follow PEP 257. Instead, NEAT follows Sphinx's info field lists as its docstring format. Please adhere to this standard as future documentation builds become more and more difficult to accurately make the more deviations are made away from this format.

In addition, Python source files identify themselves using the following header.

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
#
# Copyright (c) 2017 {{author}} ({{contact}})
# GNU GPLv3 <https://www.gnu.org/licenses/gpl-3.0.en.html>
```

Where the following apply:

`{{author}}` The initial author of the file

`{{contact}}` The contact email for the initial author of the file

We understand that this header is a pain to manually add on each commit. That is why we suggest you use a modern code editor such as Sublime Text 3 or more preferably Atom and utilize their respective file header plugins FileHeader and file-header. Please follow this standard as it makes documentation 10x easier for current and future documentation systems.

NEAT depends on Sphinx as its documentation builder. This requires the sphinx toolkit to be installed on the user's system which is extremely easy to do. By executing the following command outside of any Python virtual environments will ensure that the latest version of the Sphinx toolkit (and its dependencies) is installed and available on your system.

```
pip install sphinx
```

This dependency is also already listed in the project's `requirements.txt`.

After you have the Sphinx toolkit, documentation can be built by executing the `make html` command within the documentation directory (docs). However, changes outside of `autodoc`, which manages in-code docstrings, need to be written in reStructuredText and pointed to by the `index.rst`. For more information, simply go through Sphinx's First Steps with Sphinx.

2.1.5 Testing Conventions

NEAT tests are written using Python's standard unittest module. However, tests are executed via the nose framework.

Unittests for `neat` require both nose and codecov. These packages are not listed Tests should be run from the root directory of the repository using the following command:

```
nosetests --with-coverage
```

The `.coveragerc` file defines what folders to run tests for and what files to avoid testing.

We use a continuous integration system, TravisCI, to continually check test cases on public pushes to the GitHub repository. We also utilize codecov, which presents code coverage as reported by TravisCI after each public push. The configuration for continuous integration can be found in the standard `.travis.yml` file, found in the root of the repository.

2.1.6 Logging Conventions

Logging is enabled by default and runs on the `logging.Logger` `DEBUG` level. The default logging format is:

```
%(asctime)s - %(levelname)s - %(filename)s:%(lineno)s<%(funcName)s> %(message)s
```

The `neat` framework also comes with a custom logging exception handler which logs exceptions. All of these logging properties can be modified by changing the values of the `neat` constants:

```
import logging
import neat

# log any exceptions that occur
neat.const.log_exceptions = True

# update the logging level so just INFO and greater logs are displayed
neat.const.log_level = logging.INFO

# update the logging format so just the message is displayed
neat.const.log_format = '%(message)s'
```

Logs are stored on `stdout` as well as stored in a rotating file handler. A certain days logs are stored under the `/logs/{year}/{month}` directory in the `{month}{day}{year}.log` files. For example, the following log file path is for logs created on April 1, 2017:

```
/logs/2017/4/04012017.log
```

Log files are split every `1024 * 1024` bytes.

- Logs should primarily relay information about signal calls, and record transforms on the `DEBUG` level via `logging.debug('...')`.
- Any information about pipe connection status or general startup/shutdown information should be on the `INFO` level via `logging.info('...')`.
- Invalid input, data, configuration that doesn't cause the runtime to crash should be on the `WARNING` level via `logging.warning('...')`.
- Any invalid state or unexpected error that causes the runtime to skip over some important logic should be on the `ERROR` level via `logging.error('...')`.
- Any state causing the framework to crash should be on the `CRITICAL` level via `logging.critical('...')`.
- Finally, any caught exceptions that are used as quick fixes to errors should be logged on the `EXCEPTION` level via `log.exception('...')`.

Log lines typically also have `...` appended to the end in order to accomodate external logging parsers. This line ending is separated from the message of the log line by a space.

2.1.7 Installing Dependencies

NEAT depends on several packages provided by PyPi which need to be installed for NEAT to function correctly. These should be installed into a virtual Python environment by using the `virtualenv` package. To set this up, first install the `virtualenv` and `virtualenvwrapper` packages via pip.

```
pip install virtualenv virtualenvwrapper
```

Note, if working on Windows, it may be necessary to install the `virtualenvwrapper-win` module as well. This simply takes the functionality of `virtualenvwrapper` and translates it to batch scripts which Windows systems can run.

After installing these packages you should now have access to several scripts such as `mkvirtualenv`, `workon`, `rmvirtualenv`, and others. However, it may also be necessary to set a environmental variable to tell the installed scripts where to setup all virtual environments. This is typically done under the `WORKON_HOME` variable.

```
export WORKON_HOME=~/.virtualenvs/
```

This indicates that all virtual environments will be built and stored under the directory `~/.virtualenvs/`

NEAT is built and developed using Python 3.5+, so it may be necessary to specify the version of Python to use when creating a virtual environment.

```
mkvirtualenv --python=/usr/bin/python3 neat
```

This will create and place your current shell into the context of a new virtual environment neat (if it doesn't exist already). Note, most modern shells show an indication of what virtual environment you are currently located in. For example, a common shell prompt...

```
/home/r/Documents/Github/neat $
```

may be transformed to something resembling...

```
(neat) /home/r/Documents/Github/neat $
```

Once inside of this virtual environment it is possible to install dependencies. All of NEATs dependencies are specified in the `requirements.txt` file located in the root of the repository. This file follows pip's requirements file format. The dependencies listed in this file can be automatically installed using the virtual environment's pip script by passing the path to the requirements file after giving pip the -r flag.

```
pip install -r ./requirements.txt
```

If the pip installation goes successfully, then all listed requirements should be successfully installed to the virtual environment. To get out of the virtual environment, simply use the `deactivate` command (only available inside of a virtual environment). To re-enter a virtual environment, use the `workon neat` command, where neat is the name of the virtual environment you created.

In order for the pipes to function correctly, the servers for a pipe's database is required and must be running.

- RethinkDB for the *`RethinkDBPipe`*
- MongoDB for the *`MongoDBPipe`*

2.2 Contributing

The following subsections are for people who wish to contribute to the `neat` framework. We assume that if you want to contribute, you will abide by the standards discussed in *Developers*.

2.2.1 Issues

Best issues are a short, self contained, correct example of the problem. Providing logs for when the error occured is also very helpful.

2.2.2 Pull Requests

All pull requests must be done on the dev branch. Pull requests on the `master` branch should be ignored

2.3 Extending NEAT

The following subsections detail tasks required for extending the `neat` framework.

2.3.1 New Devices

For every new type of device that doesn't go through the Obvius, a new concrete subclass of `AbstractRequester` must be defined in order to retrieve the devices status. The amazing Requests package is provided by default in the installation of `neat` as well as BeautifulSoup and lxml for parsing XML typed content which should ease the effort of future developers. It may also (most likely) be necessary to define a new concrete subclass of `AbstractTranslator`. For each new type of device status format, a translator must be able to convert the status into a *`Record`* object for the pipes to correctly handle.

2.3.2 New Pipes

If other forms of storage are needed, a new concrete subclass of `AbstractPipe` must be defined. These typically need to handle all the logic of starting and maintaining a connection to the database (if developing a database-based pipe) and creation and deletion of databases, tables, users and potentially entries. The only thing provided to the database is a *`Record`* instance which must be deconstructed in however necessary to pass it through the pipe.

2.4 End Users

Typically end users should have to only configure the config file required by a client whose superclass is *`AbstractClient`*. For example, NEAT comes with a *`BasicClient`* which uses YAML to indicate what is required for the engine.

Starting the project can be done using a simple Python script which starts the client.

```
import neat
client = neat.BasicClient.from_config('PATH TO CONFIG')
client.start()
```

Logging and other configuration can be done by editing the constants before starting the client

```
import neat
import logging

neat.const.log_exceptions = True
neat.const.log_level = logging.DEBUG
```

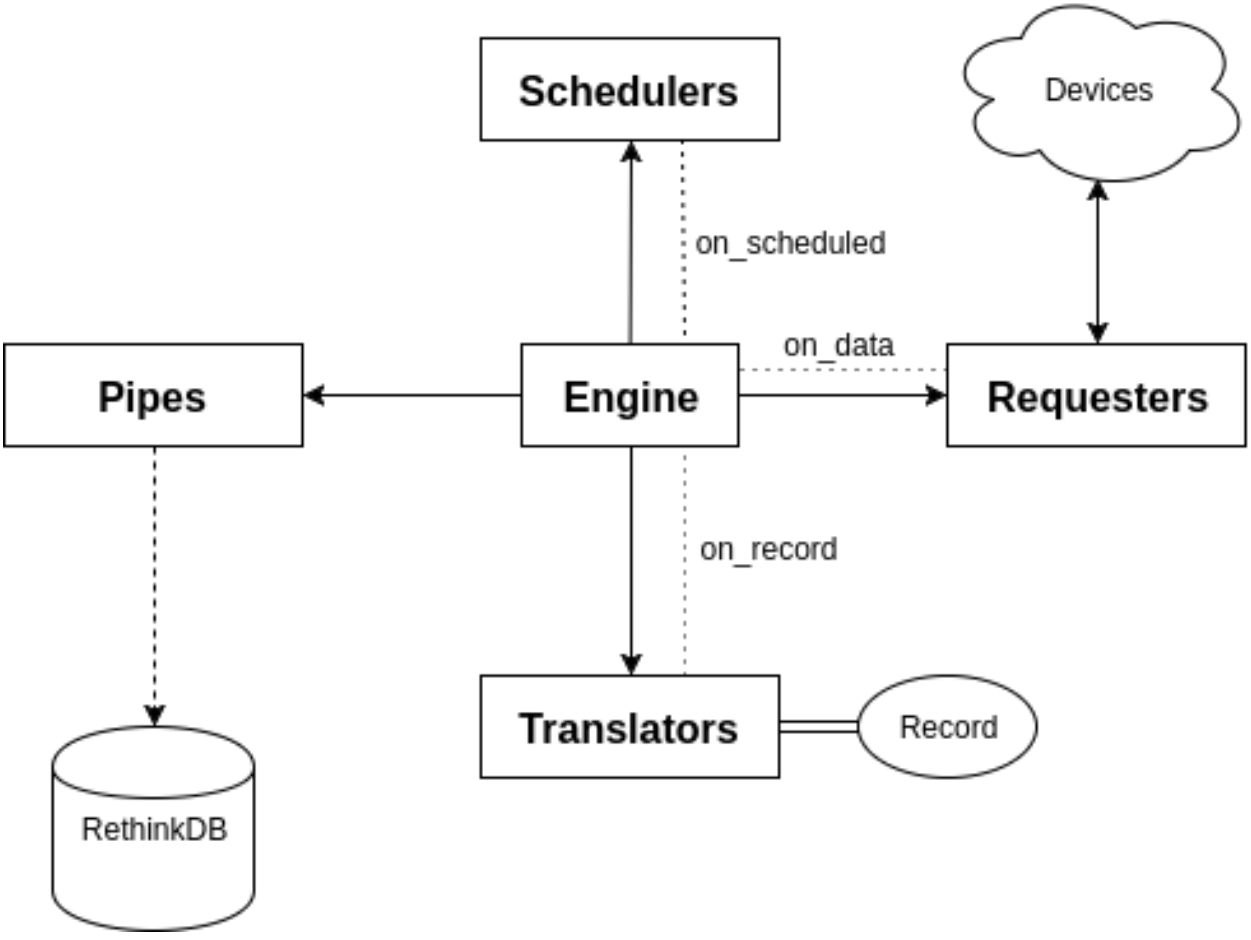
In order for pipes to function correctly, the client servers for the desired pipes must be started before running the `neat` client. This can be done by starting the RethinkDB and MongoDB pipes in a separate process like the following:

```
rethinkdb -d /path/to/rethinkdb/storage/directory
mongodb --dp-path=/path/to/mongodb/storage/directory
```

CHAPTER 3

Architecture

The NEAT project features the main communication engine as well as several *appendages* including mainly **schedulers**, **requesters**, **translators**, and **pipes**. A simple communication visualization is shown in the figure below.



These so called appendages are described within submodules of the main *neat* module as generalized in the following file structure. As you can see, the following folder structure allows for separated logic in each of the submodules while keeping connection and communication logic within the engine:

```
neat
├─ const.py
├─ client.py
├─ engine.py
├─ device.py
├─ scheduler
│   ├─ __init__.py
│   ├─ _common.py
│   ├─ simple.py
│   └─ ...
├─ requester
│   ├─ __init__.py
│   ├─ _common.py
│   ├─ obvius.py
│   └─ ...
├─ translator
│   ├─ __init__.py
│   ├─ _common.py
│   ├─ obvius.py
│   └─ ...
└─ pipes
    ├─ __init__.py
    ├─ _common.py
    ├─ rethinkdb.py
    └─ ...
```

Submodule structure mainly includes the `__init__.py` and the `_common.py` files. The `_common.py` exports an abstract class which all valid concrete classes should extend. For example in `requesters/_common.py` an abstract class `AbstractRequester` is exported which the *ObviusRequester* extends. The exported classes from the submodule should include the abstract class as well as any other concrete classes for that submodule. Because of this, concrete classes must be uniquely named and preferably have a matching suffix to their superclass. For example, as previously shown the `AbstractRequester` is the superclass for the *ObviusRequester*. The matching suffix of these two objects would in this case be *Requester*. Other submodules with abstract and concrete classes should following this convention for readability reasons.

The following sections will describe in greater detail the objectives, responsibilities, and structure of the engine as well as the previously listed submodules.

3.1 Records

The generic data model which neat produces is the *Record* object found in `models/record.py`. This object specifies the `to_dict` method which compresses the useful object information into a dictionary using the following format (a more formal jsonschema can be found in `schemas/record.json`):

```
{
  "meta": {},
  "name": "primary key unique name",
  "device_name": "human readable non-unique name",
  "type": "DEVICE_TYPE",
  "timestamp": 1234567890,
  "coord": {
```

(continues on next page)

(continued from previous page)

```
    "lon": 123.456789,
    "lat": -123.456789
  },
  "data": {
    "0": {
      "name": "unreliable-name",
      "value": 12.3456789,
      "units": "PINT_UNIT"
    }
  },
  "parsed": {
    "reliable-name": {
      "value": 12.3456789,
      "unit": "PINT_UNIT"
    }
  }
}
```

This top-level json object is built from the *Record* object in `models/record.py`. The shorter json objects in the data and parsed fields are built from the *RecordPoint* object also in `models/record.py`. It's easy to see that the record point stores information about a data point such as the name, value, and an understandable unit expression from the pint module's vanilla unit registry.

3.2 Engine

The engine's purpose is to manage communication between schedulers, requesters, translators, and pipes. It does this by hooking into the schedulers, requesters, and translators blinker signal in order to capture asynchronous output from the different running processes.

The engine should be accessed directly from the top-level module as the *Engine* class. Schedulers are mapped 1 to 1 with their scheduled requesters in the engine's private `_register` attribute on initialization of the engine. Along with this mapping the desired pipes are also passed into the engine on initialization as a list of pipe objects. Note in the following intialization example that a single *SimpleDelayScheduler* is mapped to a *ObviusRequester* for the engine's register while a single *RethinkDBPipe* is given engine.

```
import neat
engine = neat.Engine({
    neat.scheduler.SimpleDelayScheduler(...):
    neat.requester.ObviusRequester(...)
}, pipes=[neat.pipe.RethinkDBPipe(...)])
```

The engine's logic flow works as the following:

1. Schedulers are started as their own child processes of the engine
2. A scheduler communicates over its signal when its requester should run
3. Engine intercepts the scheduler's signal with the `on_scheduled` method
4. Engine determines what requester should run and calls the *request()* method
5. A requester communicates over its signal when it receives data
6. Engine intercepts the requester's signal with the `on_data` method
7. Engine determines which translator is *capable* of translating the received data and calls the `translate` method
8. A translator communicates over its signal when the *Record* model has been built successfully

9. Engine intercepts the translator's signal with the `on_record` method
10. Engine throws the record into each of the valid pipes via the *`accept()`* method
11. Pipes handle any necessary storage logic

3.3 Schedulers

The purpose of a scheduler is to provide a way of telling the engine when a requester should be called. Because these schedulers must execute with their own specific time-frames they are subclasses of `AbstractScheduler` which itself is a subclass of `multiprocessing.Process` allowing these schedulers to be run as children processes of the process containing neat's engine. The `AbstractScheduler` provides an anonymous blinker signal attribute and requires that concrete classes implement a *`run()`* method which starts (most likely) an infinite loop of request scheduling logic.

Although new schedulers may need to take into account device specific refresh rates or communication rules, most of the time the best option is to use the already provided *`SimpleDelayScheduler`* from `scheduler/simple.py` which employs a delay by sleeping the process for a specified second delay.

Note: Because schedulers are subclasses of `multiprocessing.Process` if an `__init__` method is required of a concrete scheduler, the superclass's `__init__` must be called before any attribute assignment.

For example, the *`SimpleDelayScheduler`* requires an input parameter to specify the second delay which should be used. The following simplified class snippet was used:

```
class SimpleDelayScheduler(AbstractScheduler):

    def __init__(self, delay: float=1.0):
        super().__init__()
        self.delay = delay
```

3.4 Requesters

The purpose of a requester is to ensure that some device's state is retrieved and passed back to the engine. As opposed to schedulers, requesters are not their own spawned processes, instead they run alongside the engine when triggered from the `on_scheduled` signal.

Concrete requesters must extend from `AbstractRequester` which also provides an abstract blinker signal and requires that the requester implements a method *`request()`* which sends some request to a device for current status. In order to keep blocking to a minimum, requesters utilize the requests module and specify request hooks to be most optimal in not blocking engine execution. Once the data has been retrieved the requester instance as well as the retrieved data and any additional named parameters to the requester's initialization is sent back over the requesters signal which can then be caught by the engine. These additional parameters are typically *`Record`* fields that need to be user-specified due to the device not containing that information. An example of this is typically the longitude and latitude of the device since many devices do not keep track of that information.

Take the following requester initialization for example:

```
requester = neat.requester.ObviusRequester(
    obvius_ip='123.123.123.123',
    obvius_port=80,
    obvius_user='SOMEUSER',
```

(continues on next page)

(continued from previous page)

```
    obvius_pass='SOMEPASS',
    name='DEVICE_NAME',
    type='DEVICE_TYPE',
    lat=123.4567890,
    lon=123.4567890
)
```

In this instance, although *ObviusRequester* cannot handle `lat` and `lon` in requester initialization, it still requires those fields in order for the translator to have those fields handy when building the *Record*. Therefore, the extraneous fields which cannot be used in initialization for the requester are included in the signal along with the data and the requester instance.

3.5 Translators

The purpose of a translator is to provide a simple interface to create a *Record* object from some data retrieved by a requester. A single given translator may be acceptable for translating multiple formats of data. This is specified in the `supported_requesters` attribute of a concrete translator as a list of string class names of the supported requesters.

Note: The current method of translator discovery is *naive* as it returns the first translator is sees which specifies that it can handle data from a specific requester. This process can be seen in `translators/__init__.py` as *get_translator()*.

Valid concrete translators must extend from `AbstractTranslator` as usual. `AbstractTranslator` provides an anonymous blinker signal and requires a *translate()* method for synchronously creating and sending the built *Record* object over the provided signal.

Note the engine lazily instantiates the translators only when they are required. Therefore, initialization parameters to concrete translators is currently not supported in the neat engine.

3.6 Device Types

The purpose of a device type is to ensure that the data comming in from multiple different types of devices from multiple requesters can have their points generalized into the `parsed` field of a *Record*. The allowed device types are stored in the `device.py` and are encapsulated within the *DeviceType* enumeration along with a unique hexadecimal id and an instance to the device. Correct parsing of the data fields currently relies on the `parsed` fields contained within the `config.yml`. With the addition of new device types and different requesters that do no utilize the Obvius' device points, it may be neccessary to change the logic of the *parse()* function.

The *parse()* function takes the populated data fields along with the `parsed` config configuration to determine what attributes of the record's `data` to load and convert to a uniform pint unit. This information is the placed within the `parsed` dictionary of the *Record* which can then be serialized for the pipe's usage.

3.7 Pipes

The purpose of a pipe is to provide any and all logic for handling the storage created records into various different formats. The provided concrete pipe is a *RethinkDBPipe* which places records into a rethinkdb database as they come in.

Valid pipes must extend from `AbstractPipe` which provides an anonymous blinker signal and requires that the pipe have an *`accept()`* method which accepts a single *`Record`* object. Once a record has been successfully committed to wherever it needs to be, the pipe must send itself and the record over the provided signal where the engine can intercept the signal in the `on_complete` signal.

CHAPTER 4

NEAT Package

4.1 Module contents

4.2 Subpackages

4.2.1 neat.models package

Module contents

Submodules

neat.models.record module

class neat.models.record.**Record**(***kwargs*)

Bases: neat.models._common.AbstractModel

A model representation of a record.

data
The device's raw data points.

device_name
The human readable name of the device.

lat
The latitude of the device.

lon
The longitude of the device.

name
The primary name of the device.

parsed
The device's parsed data points.

timestamp
The record's creation unix timestamp.

to_dict() → dict
Builds a serializable representation of the record.

Returns A serializable representation of the record

Return type dict

ttl
The record's time to live in seconds.

type
The type of the device.

validate() → bool
Self validates the record.

Returns True if valid, otherwise False

Return type bool

class neat.models.record.**RecordPoint**(***kwargs*)
Bases: object

A record point representation.

Note: Not a subclass of neat.models._common.AbstractModel

name
The name of the record point.

number
The number of the record point.

to_dict() → dict
Builds a serializable representation of the record point.

Returns A serializable representation of the record point

Return type dict

unit
The pint unit expression of the record point.

value
The value of the record point.

4.2.2 neat.scheduler package

Module contents

Submodules

neat.scheduler.simple module

class neat.scheduler.simple.**SimpleDelayScheduler**(*delay: float = 1.0*)
Bases: neat.scheduler._common.AbstractScheduler

The scheduler for simple requesters.

Note: Required, that all subclasses call super initialization

---

delay
The delay period in between scheduled requests.

run() → None
Starts the infinite loop for signaling scheduled requests.

Returns Does not return

Return type None

4.2.3 neat.requester package

Module contents

Submodules

neat.requester.obvius module

class neat.requester.obvius.**ObviusRequester**(*device_id: int*, *obvius_ip: str*, *obvius_user: str*, *obvius_pass: str*, *obvius_port: int = 80*, *timeout: int = 10*, ***kwargs*)
Bases: neat.requester._common.AbstractRequester

The requester for the Obvius server.

receive(*resp: requests.models.Response*, **args*, ***kwargs*) → None
The receiver of information from the requester.

Parameters

- **resp** (*requests.Response*) – The response of the request
- **args** (*list*) – Extra arguments of the request
- **kwargs** (*dict*) – Extra named arguments of the request

Returns Does not return

Return type None

request() → None
Request information from the obvius.

Returns Does not return

Return type None

4.2.4 neat.translator package

Module contents

neat.translator.**get_translator**(*requester_name: str*) → neat.translator._common.AbstractTranslator
Tries to retrieve a supported translator given a requesters name.

Parameters **requester_name** (*str*) – The requester's class name

Returns A supported translator

Return type AbstractTranslator

Submodules

neat.translator.obvius module

class neat.translator.obvius.**ObviusTranslator**
Bases: neat.translator._common.AbstractTranslator

The translator for Obvius devices.

parser
The *xml* parser to use for parsing the returned requester content.

supported_requesters = ('ObviusRequester',)

translate(*data: str*, *meta: dict = {}*) → None
Translates Obvius data to a record.

Parameters

- **data** (*str*) – The xml returned from the Obvius endpoint
- **meta** (*dict*) – Any additional data given to the requester

Returns Does not return

Return type None

unit_map
The mapping of Obvius units to valid pint units.

validate(*data: str*) → bool
Checks if the data from the Obvius is valid.

Parameters **data** (*str*) – The data returned from a supported requester

Returns True if the data is valid, otherwise False

Return type bool

4.2.5 neat.pipe package

Module contents

Submodules

neat.pipe.mongodb module

class neat.pipe.mongodb.**MongoDBPipe**(*ip: str*, *port: int*, *table: str*, *entry_delay: int = 600*)
Bases: neat.pipe._common.AbstractPipe

A record pipe for MongoDB.

Note: Records are always placed in the *neat* table.

---

accept(*record: neat.models.record.Record*) → None
Accepts a record to be placed into the MongoDB instance.

Parameters **record** (Record) – The record to be placed in the MongoDB instance

Returns Does not return

Return type None

client
The client attached to the MongoDB uri.

> **Warning:** MongoDB driver connections are not fork safe

db
The database of the client to write to.

table
The table of the database to write to.

validate() → bool
Self validates the MongoDB pipe.

Returns True if the pipe is valid, otherwise False

Return type bool

neat.pipe.rethinkdb module

class neat.pipe.rethinkdb.**RethinkDBPipe**(*ip: str*, *port: int*, *table: str*, *clean_delay: int = 300*)
Bases: neat.pipe._common.AbstractPipe

A record pipe for RethinkDB.

Note: Records are always placed in the *neat* table.

---

accept(*record: neat.models.record.Record*) → None
Accepts a record to be placed into the RethinkDB instance.

Parameters **record** (Record) – The record to be placed into the RethinkDB instance

Returns Does not return

Return type None

clean() → None
Cleans dead records from the RethinkDB instance.

Returns Does not return

Return type None

connection
The client attached to the RethinkDB uri.

> **Warning:** RethinkDB driver connections are not thread safe

validate() → bool
Self validates the RethinkDB pipe.

Returns True if the pipe is valid, otherwise False

Return type bool

4.3 Submodules

4.4 neat.const module

Module constants object.

exception neat.const.**ModuleConstantException**(*message: str*, *code: int = None*)
Bases: Exception

Custom exception for constants namespace.

4.5 neat.client module

class neat.client.**AbstractClient**
Bases: object

The basic class for all valid clients.

static from_config(*config_path: str*)
Creates a client from a config file.

Parameters **config** (*str*) – The path of the config file to load from

Returns An instance of the created client

Return type *AbstractClient*

start() → None
Starts the engine.

Returns Does not return

Return type None

class neat.client.**BasicClient**(*config: dict*)
Bases: *neat.client.AbstractClient*

A very basic client for engine initalization.

static from_config(*config: str*)
Creates a BasicClient from a config file.

Parameters **config** (*str*) – The path of the config file to load from

Returns An instance of the created BasicClient

Return type *BasicClient*

start() → None
Starts the engine.

Returns Does not return

Return type None

4.6 neat.engine module

class neat.engine.**Engine**(*register: Dict[neat.scheduler._common.AbstractScheduler, neat.requester._common.AbstractRequester] = {}, pipes: List[neat.pipe._common.AbstractPipe] = []*)
Bases: object

Provides communication between all of the subpackages.

on_commit(*piper: neat.pipe._common.AbstractPipe*, *record: neat.models.record.Record*) → None
Event handler for when pipes finish writing out a record.

Parameters

- **piper** (*AbstractPipe*) – The pipe who wrote the record out
- **record** (Record) – The record that was written

Returns Does not return

Return type None

on_data(*requester: neat.requester._common.AbstractRequester*, *data: str*, *meta: dict*) → None
Event handler for when requesters get a response from their device.

Parameters

- **requester** (*AbstractRequester*) – The requester who retrieved the data
- **data** (*str*) – The data returned from the device
- **meta** (*dict*) – Any additional fields required to properly interpret data

Returns Does not return

Return type None

on_record(*record: neat.models.record.Record*) → None
Event handler for when translators finish translation of some data.

Parameters **record** (Record) – The translated record

Returns Does not return

Return type None

on_scheduled(*scheduler: neat.scheduler._common.AbstractScheduler*) → None
Event handler for when schedulers trigger their mapped requesters.

Parameters **scheduler** (*AbstractScheduler*) – The scheduler that needs to run its requester

Returns Does not return

Return type None

on_start = <blinker.base.Signal object>

on_stop = <blinker.base.Signal object>

pipes
The list of pipe objects that are handling created records.

register
The mapping of schedulers to requesters.

start() → None
Starts the engine.

Returns Does not return

Return type None

stop() → None
Stops the engine.

Returns Does not return

Return type None

translators
The list of translator objects that have been needed.

4.7 neat.device module

class neat.device.**AbstractDevice**
Bases: object

The base class for all valid devices.

fields
The expected device fields.

name
The name of the device.

parse(*record: neat.models.record.Record*) → Dict[str, neat.models.record.RecordPoint]
Parses a given record for the necessary device fields.

Parameters **record** (Record) – The record to parse

Returns A dictionary of field names mapped to record points

Return type dict

ureg
The unit registry for all devices.

class neat.device.**DeviceType**
Bases: enum.Enum

An enumeration of available device types.

Note: Maps types to instances not classes

ENERGY = (5, <neat.device.EnergyDevice object>)

HVAC = (2, <neat.device.HVACDevice object>)

PV = (1, <neat.device.PVDevice object>)

SOLAR_THERM = (3, <neat.device.SolarThermalDevice object>)

TEMP = (6, <neat.device.TemperatureDevice object>)

UNKNOWN = (0, <neat.device.UnknownDevice object>)

WIND = (4, <neat.device.WindDevice object>)

class neat.device.**EnergyDevice**
Bases: *neat.device.AbstractDevice*

Defines a generic energy device.

fields = {}

name = 'Energy Device'

class neat.device.**HVACDevice**
Bases: *neat.device.AbstractDevice*

Defines a heating, ventilation, and cooling device.

fields = {}

name = 'HVAC Device'

class neat.device.**PVDevice**
Bases: *neat.device.AbstractDevice*

Defines a photovoltaic device.

fields = {}

name = 'PV Device'

class neat.device.**SolarThermalDevice**
Bases: *neat.device.AbstractDevice*

Defines a solar thermal device.

fields = {'energy_rate': 'btu / hour', 'energy_total': 'megabtu', 'flow_rate': 'gal

name = 'Solar Thermal Device'

class neat.device.**TemperatureDevice**
Bases: *neat.device.AbstractDevice*

Defines a generic temperature device.

fields = {}

name = 'Temperature Device'

class neat.device.**UnknownDevice**
Bases: *neat.device.AbstractDevice*

Defines an unknown device type.

Note: Primarily used for Obvius virtual meters

---

fields = {}

name = 'Unknown Device'

class neat.device.**WindDevice**

Bases: *neat.device.AbstractDevice*

Defines a wind based device.

fields = {'inverter_energy_total': 'kilowatthour', 'inverter_real': 'kilowatt', 'rot

name = 'Wind Device'

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

O

P

R

S

T

U

V

W