Unide Python Documentation

Release 0.1.0

Eclipse Unide

Jan 23, 2018

Contents

CHAPTER 1

Unide Python

This Python package is part of the Eclipse Unide Project and provides an API for generating, parsing and validating PPMP payloads. PPMP, the "Production Performance Management Protocol" is a simple, JSON-based protocol for message payloads in (Industrial) Internet of Things applications defined by the Eclipse IoT Working Group. Implementations for other programming languages are available from the Unide web site.

The focus of the Python implementation is ease of use for backend implementations, tools and for prototyping PPMP applications. Generating a simple payload and sending it over MQTT using Eclipse Paho is a matter of just a few lines:

```
>>> import unide
>>> import paho.mqtt.client as mqtt
>>> client = mqtt.Client()
>>> client.connect("localhost", 1883, 60)
>>> device = unide.Device("Devive-001")
>>> measurement = device.measurement(temperature=36.7)
>>> client.publish(topic="sample", measurement)
```

1.1 Installation

The latest version is available in the Python Package Index (PyPI) and can be installed using:

```
pip install unide-python
```

`unide-python` can be used with Python 2.7, 3.4, 3.5 and 3.6.

Source code, including examples and tests, is available on GitHub: https://github.com/eclipse/unide.python

To install the package from source:

```
git clone git@github.com:eclipse/unide.python.git
cd unide.python
python setup.py install
```

1.2 Contributing

This is a straightforward Python project, using *setuptools* and the standard `setup.py` mechanism. You can run the test suite using `setup.py`:

```
python setup.py test
```

There also is a top-level `Makefile` that builds a development environment and can run a couple of developer tasks. We aim for 100% test coverage and use tox to test against all supported Python releases. To run all tests against all supported Python versions, build the documentation locally and an installable wheel, you'll require pyenv and a decent implementation of make. `make all` will create a virtualenv `env` in the project directory and install the necessary tools (see `tools.txt`).

For bug reports, suggestions and questions, simply open an issue in the Github issue tracker. We welcome pull requests.

1.3 Documentation

Detailed documentation is available on Read the Docs: http://unidepython.readthedocs.io/en/latest/.

CHAPTER 2

Programming Guide

PPMP is simple enough to be reasonably used from Python without an API at all. The simplest possible PPMP measurement payload, transmitting just one sensor reading for "temperature", looks like this:

```
{
  "content-spec":
    "urn:spec://eclipse.org/unide/measurement-message#v2",
  "device": {
    "deviceID": "a4927dad-58d4-4580-b460-79cefd56775b"
  },
  "measurements": [{
    "ts": "2002-05-30T09:30:10.123+02:00",
    "series": {
       "$_time": [ 0 ],
       "temperature": [ 45.4231 ]
  }
}
```

The main use cases for *unide* are handling and generating complex payloads programmatically, and parsing and validating incoming PPMP messages. *unide* is suitable for backend implementations receiving PPMP data, it can run on gateways supporting Python, and it is useful for quickly scripting PPMP applications and tools.

2.1 Getting Started

unide provides a Python class for every entity described in the PPMP specification. Classes have read-write attributes for each property in the specification. All properties can be passed directly into the class constructor using positional and named arguments.

Unset properties are *None* in the Python API, but will not be serialized as 'null' into JSON, i.e. unset properties will not appear in the JSON output at all. Strings are mapped to and from Python Unicode strings (i.e. *unicode* for Python 2, and *str* for Python 3). Numeric values are mapped to Python *float*. Timestamps are mapped to Python's `datetime` (see *Timestamps* for details).

Every PPMP entity can be build separately, and re-used later to assemble a complete payload. A central entity in PPMP is the *Device*, that has just one mandatory property, its *deviceID*:

```
>>> from unide.common import Device
>>> device = Device("Device-001")
```

```
>>> print(device.deviceID == "Device-001")
Device-001
```

All other properties of `device` are now *None* and can be assigned a value:

```
>>> print(device.operationalStatus)
None
>>> device.operationalStatus = "running"
>>> print(device.operationalStatus)
running
```

PPMP objects can be printed:

```
>>> print(device)
Device(deviceID=Device-001, operationalStatus=running)
```

In PPMP, all messages originate from a device. The *Device* class therefore has convenience APIs to quickly produce complete payloads. The example below produces a simple *MeasurementPayload* using `Device.measurement()`:

```
>>> msg = device.measurement(temperature=36.7)
>>> print(msg)
{"device": {"deviceID": "Device-001", "operationalStatus": "running"}, "content-
↪spec": "urn:spec://eclipse.org/unide/measurement-message#v2", "measurements": [{
↪"ts": "2017-09-13T22:23:26.840407+02:00", "series": {"temperature": [36.7], "$_
↪time": [0]}}]}
```

The other two types of PPMP messages are *MessagePayload* and *ProcessPayload* and can be produced using `Device.message()` and `Device.process()` respectively.

We can create the same message using the lower-level APIs by building each component separately. To do that, we have to create a `Series` object and explicitly declare the *dimension* `temperature` that we want to provide:

```
>>> from unide.measurement Series
>>> series = Series("temperature")
>>> series.add_sample(0, temperature=36.7)
```

Then, we create a `Measurement` object and assemble a `MeasurementPayload` using the components we've just created:

```
>>> from unide.measurement import Measurement, MeasurementPayload
>>> from unide import util
>>> m = Measurement(ts=util.local_now(), series=series)
>>> payload = MeasurementPayload(device=device)
>>> payload.measurements.append(m)
```

The *measurements* property of the payload object is just a normal Python list of *Measurement* objects.

Finally, `payload` can be converted to JSON by using `dumps()` from `unide.util`. The string returned by *dumps* can be send as a payload using a transport protocol like HTTP/REST or MQTT. *unide* by itself does not implement any transport protocol:

```
>>> from unide.util import loads
>>> print(dumps(payload, indent=4))
{
    "device": {
        "deviceID": "Device-001"
    },
    "content-spec": "urn:spec://eclipse.org/unide/measurement-message#v2",
    "measurements": [
        {
            "ts": "2017-09-13T23:40:46.685521+02:00",
```

```
            "series": {
                "$_time": []
            }
        }
    ]
}
```

2.2 Validation and Parsing

The *unide* APIs validate inputs. For example, the maximum length for device identifiers is 36. Trying to assign a longer id raises a *ValueError* exception:

```
>>> device = Device("PPMP HAS A SIZE RESTRICTION FOR DEVICE IDs!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "measurement.py", line 225, in __init__
    self.deviceID = deviceID
  File "schema.py", line 96, in set
    value = check(self, name, value, constraint)
  File "schema.py", line 84, in check
    .format(name=name, value=value, classname=type(self).__name__))
ValueError: u'PPMP HAS A SIZE RESTRICTION FOR DEVICE IDs!' is not an appropriate␣
↪value for 'Device.deviceID'
```

Parsing a PPMP message is done using `loads()`:

```
>>> from unide.util import loads
>>> msg = loads(open("tests/message.json").read())
>>> print(msg)
MessagePayload(device=Device(operationalStatus=normal, deviceID=2ca5158b-8350-4592-
↪bff9-755194497d4e, metaData={u'swVersion': u'2.0.3.13', u'swBuildID': u'41535'}),
↪ messages=[<unide.message.Message object at 0x1095938d0>, <unide.message.Message␣
↪object at 0x109af6510>], content-spec=urn:spec://eclipse.org/unide/machine-
↪message#v2)
```

`loads()` automatically detects the payload type and returns the appropriate *unide* object. If the payload type can not be detected, an exception will be raised.

Besides trying to detect the PPMP type, parsed messages will *not* be validated by default. Malformed messages can be parsed, and all recognizable information can be accessed. A message can be validated using `problems()` after loading it:

```
>>> msg = loads(open("tests/invalid.json").read())
>>> msg.problems()
[u"'xdevice' is not a valid key for 'MessagePayload' objects"]
```

`problems()` returns a list of issues. An empty list indicates a valid payload.

To validate a payload while parsing it, one can set the `validate` flag for *loads*. When the payload is not valid, a *ValidationError* exception is raised:

```
>>> msg = loads(open("tests/invalid.json").read(), validate=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/frank/Projects/unide/cslab/unide.python/src/unide/util.py", line 51,
↪ in loads
    raise ValidationError(errors)
  unide.util.ValidationError: 'xdevice' is not a valid key for 'MessagePayload'␣
↪objects
```

2.3 Timestamps

All PPMP messages carry one or more timestamps. Timestamps are represented by *unide* as Python *datetime.datetime* objects. In Python, *datetime* objects come in two flavours: "naive" – without timezone information, and "aware" – including timezone information. While the PPMP specification is not explicit about this, *unide* automatically makes all timestamps "aware". If you assign a "naive" *datetime* to a PPMP property, it will be made "aware" by adding the local timezone offset:

```
>>> from unide.measurement import Measurement
>>> import datetime
>>> now = datetime.datetime.now()
>>> m = Measurement(ts=now)
>>> print(now)
2017-09-13 22:56:59.329554
>>> print(m.ts)
2017-09-13 22:56:59.329554+02:00
>>>
```

Note the difference! "Naive" and "aware" timestamps are not even compatible in Python:

```
>>> now == m.ts
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can't compare offset-naive and offset-aware datetimes
```

We therefore recommend to always use "aware" *datetime* objects to avoid awe and confusion.

unide provides two functions in its `unide.util` module to help with that: `local_now()` computes the timestamp for the current time including the local timezone offset, and `local_timezone(value)` converts any naive *datetime* to "aware" using the offset of the local timezone.

CHAPTER 3

API Reference

3.1 unide.common

Schema objects commonly used by more than one PPMP payload type.

3.2 PPMP Measurements

3.3 PPMP Messages

3.4 PPMP Process

3.5 Schema

3.6 Utilities

CHAPTER 4

Indices and tables

- genindex
- modindex
- search