omi

Release 0.0.2

Jul 18, 2019

Contents

CHAPTER 1

Overview

docs	
tests	
package	

A library to process and translate open energy metadata.

- Free software: AGPL-3.0

1.1 Installation

```
pip install omi
```

1.2 Documentation

https://omi.readthedocs.io/

1.3 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 2

Installation

At the command line:

```
pip install omi
```

CHAPTER 3

Usage

3.1 omi

```
omi [OPTIONS] COMMAND [ARGS]...
```

3.1.1 translate

```
omi translate [OPTIONS] FILE_PATH
```

Options

-f <f>
: Dialect identifier of the input

-t <t>
: Dialect identifier to translate to

-o <o>
: Output file

Arguments

FILE_PATH
: Required argument

A list of available dialects can be found in the *dialect segment*

To use omi in a project:

```
import omi
```

CHAPTER 4

Reference

4.1 omi

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

omi could always use more documentation, whether as part of the official omi docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/OpenEnergyPlatform/omi/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *omi* for local development:

1. Fork omi (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/omi.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Martin Glauer - openenergy-platform.org

CHAPTER 7

Changelog

7.1 0.0.0 (2019-01-28)

- First release on PyPI.

CHAPTER 8

OMI Metadata

class omi.structure.**Compilable**

An abstract class for all metadata components.

__compiler_name__ = None

Used to identify the appropriate compiler function for this structure

CHAPTER 9

Dialects

This section discusses the concepts of *Parser*, *Compiler* and *Dialect*

The OMI tool handles all metadata in an internal data structure that covers the relevant information needed to describe data. Different metadata formats (e.g. the OEP metadata format) can be **parsed** into this structure or **compiled** from it.

Therefore, OMI uses the notion of **Parser** and **Compiler**. A *Dialect* combines the functionalities of *Parser*, *Compiler* and adds some convenience methods to it. Each dialect has an id that can be used to call it via the *command line interface*

Available dialects are:

- *oep-v1.3*
- *oep-v1.4*
- *oep-rdf-v1.4*

class omi.dialects.base.dialect.**Parser**

A parser is used to transform to read a specific metadata format and transform it into the internal metadata representation.

is_file_valid(*file: str*, ***kwargs*)

Verify whether the contents of the file under *file* is parsable by this parser

Parameters

- **file** (*str*) – Path to the file to validate
- ****kwargs**

Returns *bool* – Returns *True* iff the file's content is parsable

is_valid(*inp: str*) → bool

Verify whether *inp* is a sting representation that is parsable by this parser

Parameters **inp** (*str*) – String to verify

Returns *bool* – Indicated whether this object is parsable or not

load_string(*string: str*, **args*, ***kwargs*)
Load a string into the structure represented by the dialect :Parameters: **string** (*str*)

Returns *Translates the passed string into the format used as input for this parser*

parse(*structure: T*, **args*, ***kwargs*) → omi.structure.OEPMetadata
Transforms the input structure into metadata as used by the OpenEnergyPlatform

Parameters **inp** (*str*) – The input string that should be parsed into OEP metadata

Returns *OEPMetadata* – OEP metadata represented by *inp*

parse_from_string(*string: str*, *load_args=None*, *parse_args=None*, *load_kwargs=None*, *parse_kwargs=None*) → omi.structure.OEPMetadata
Parse a string into `OEPMetadata`

Parameters **string**

class `omi.dialects.base.dialect.`**Compiler**
Compiles *`Compilable`* objects into the respective metadata format. Every omi compiler should inherit from this class

visit(*obj*, **args*, ***kwargs*)
Calls the respective compiler for *`Compilable`* objects respective to `Compilable.__compiler_name__`

Parameters **obj** – Object to compile

Returns Metadata representation of *obj*

class `omi.dialects.base.dialect.`**Dialect**

compile(*obj: omi.structure.OEPMetadata*, **args*, ***kwargs*)
Compiles the passed `OEPMetadata`-object into the structure fitting for this dialect

Parameters **obj** – The `OEPMetadata`-object to compile

compile_and_render(*obj: omi.structure.OEPMetadata*, **args*, ***kwargs*)
Combination of *`compile()`* and *`render()`*.

parse(*string: str*, **args*, ***kwargs*) → omi.structure.OEPMetadata
Loads the passed string into an `OEPMetadata`-object.

Parameters **string** – The string to parse

Returns The `OEPMetadata`-object represented by *string*

render(*structure*, **args*, ***kwargs*) → str
Turns the structure used by this dialect into a string

Parameters **structure** – The structure to stringify

Returns A string representation of *structure*

CHAPTER 10

Translations in Python

In order to perform the translation from one dialect to another, you need to parse your input using the respective input dialect. As a minimal example, let's say you have a metadata string in the outdated version 1.3 and aim to update it to the more modern version 1.4.

10.1 Step 1: Parse it

Your first step is to parse the given string using `omi.dialects.oep.dialect.OEP_V_1_3_Dialect`. For starters, we use the most basic metadata string: The empty dictionary

```
>>> inp = '{}'
>>> dialect1_3 = OEP_V_1_3_Dialect()
>>> parsed = dialect1_3.parse(inp)
>>> parsed
OEPMetadata(name=None,title=None,identifier=None,description=None,languages=None,
↪keywords=None,publication_date=None,context=None,spatial=None,temporal=None,
↪sources=None,license=None,contributions=None,resources=None,review=None,
↪comment=None)
```

The input has been parsed into the internal structure i.e. an `OEPMetadata`-object.

10.2 Step 2: Change it

If needed you can feel free to manipulate this string according to your use case. Don't forget to document your changes under contributions (ToDo) ;)

In this example, we will add an identifier, as is required by OEP-Metadata v1.4

```
>>> parsed.identifier = "unique_id"
>>> parsed
OEPMetadata(name=None,title=None,identifier=unique_id,description=None,languages=None,
↪keywords=None,publication_date=None,context=None,spatial=None,temporal=None,
↪sources=None,license=None,contributions=None,resources=None,review=None,
↪comment=None)
```

(continues on next page)

(continued from previous page)

10.3 Step 3: Compile it

Now that we have an `OEPMetadata`-object we are happy with, we want to translate it to the new metadata format by using the respective dialect

```
>>> dialect1_4 = OEP_V_1_4_Dialect()
>>> dialect1_4.compile(parsed)
OrderedDict([('name', None), ('title', None), ('id', 'unique_id'), ('description',␣
↪None), ('language', None), ('keywords', None), ('publicationDate', None), ('context
↪', None), ('spatial', None), ('temporal', None), ('sources', None), ('licenses',␣
↪None), ('contributors', None), ('resources', None), ('review', None), ('metaMetadata
↪', OrderedDict([('metadataVersion', 'OEP-1.4'), ('metadataLicense', OrderedDict([(
↪'name', 'CC0-1.0'), ('title', 'Creative Commons Zero v1.0 Universal'), ('path',
↪'https://creativecommons.org/publicdomain/zero/1.0/')]))])), ('_comment', None)])
```

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

C

D

F

I

L

O

P

R

V