calamus

Release 0.1.1

Swiss Data Science Center <contact@datascience.ch>

Feb 28, 2023

CONTENTS:

1 Installation **3**

2 Usage **5**
- 2.1 Declare schemes . . . 5
- 2.2 Serializing objects ("Dumping") . . . 5
- 2.3 Deserializing objects ("Loading") . . . 6
- 2.4 Validation of properties in a namespace using an OWL ontology . . . 6
- 2.5 Annotations . . . 8

3 Support **9**
- 3.1 calamus API . . . 9
- 3.2 License . . . 13
- 3.3 Contributing . . . 13
- 3.4 Changes . . . 16

4 Indices and tables **21**

Python Module Index **23**

Index **25**

calamus is a library built on top of marshmallow to allow (de-)Serialization of Python classes to JSON-LD

CHAPTER

ONE

INSTALLATION

calamus releases and development versions are available from PyPI. You can install it using any tool that knows how to handle PyPI packages.

With pip:

```
$ pip install calamus
```

CHAPTER

TWO

USAGE

Assuming you have a class like

```
class Book:
    def __init__(self, _id, name):
        self._id = _id
        self.name = name
```

2.1 Declare schemes

You can declare a schema for serialization like

```
from calamus import fields
from calamus.schema import JsonLDSchema

schema = fields.Namespace("http://schema.org/")

class BookSchema(JsonLDSchema):
    _id = fields.Id()
    name = fields.String(schema.name)

    class Meta:
        rdf_type = schema.Book
        model = Book
```

The `fields.Namespace` class represents an ontology namespace.

Make sure to set `rdf_type` to the RDF triple type you want get and `model` to the python class this schema applies to.

2.2 Serializing objects ("Dumping")

You can now easily serialize python classes to JSON-LD

```
book = Book(_id="http://example.com/books/1", name="Ilias")
jsonld_dict = BookSchema().dump(book)
#{
#    "@id": "http://example.com/books/1",
#    "@type": "http://schema.org/Book",
#    "http://schema.org/name": "Ilias",
```

(continues on next page)

(continued from previous page)

```
#}

jsonld_string = BookSchema().dumps(book)
#'{"@id": "http://example.com/books/1", "http://schema.org/name": "Ilias", "@type":
↪"http://schema.org/Book"}')
```

2.3 Deserializing objects ("Loading")

You can also easily deserialize JSON-LD to python objects

```
data = {
    "@id": "http://example.com/books/1",
    "@type": "http://schema.org/Book",
    "http://schema.org/name": "Ilias",
}
book = BookSchema().load(data)
#<Book(_id="http://example.com/books/1", name="Ilias")>
```

2.4 Validation of properties in a namespace using an OWL ontology

You can validate properties in a python class during serialization using an OWL ontology. The ontology used in the example below doesn't have `publishedYear` defined as a property.

```
class Book:
    def __init__(self, _id, name, author, publishedYear):
        self._id = _id
        self.name = name
        self.author = author
        self.publishedYear = publishedYear

class BookSchema(JsonLDSchema):
    _id = fields.Id()
    name = fields.String(schema.name)
    author = fields.String(schema.author)
    publishedYear = fields.Integer(schema.publishedYear)

    class Meta:
       rdf_type = schema.Book
       model = Book

book = Book(id="http://example.com/books/2", name="Outliers", author="Malcolm Gladwell
↪", publishedYear=2008)

data = {
    "@id": "http://example.com/books/3",
    "@type": "http://schema.org/Book",
    "http://schema.org/name" : "Harry Potter & The Prisoner of Azkaban",
    "http://schema.org/author" : "J. K. Rowling",
    "http://schema.org/publishedYear" : 1999
}
```

(continues on next page)

(continued from previous page)

```
valid_invalid_dict = BookSchema().validate_properties(
    data,
    "tests/fixtures/book_ontology.owl"
)
# The ontology doesn't have a publishedYear property
# {'valid': {'http://schema.org/author', 'http://schema.org/name'}, 'invalid': {
↪'http://schema.org/publishedYear'}}

validated_json = BookSchema().validate_properties(book, "tests/fixtures/book_ontology.
↪owl", return_valid_data=True)
#{'@id': 'http://example.com/books/2', '@type': ['http://schema.org/Book'], 'http://
↪schema.org/name': 'Outliers', 'http://schema.org/author': 'Malcolm Gladwell'}
```

You can also use this during deserialization.

```
class Book:
    def __init__(self, _id, name, author):
        self._id = _id
        self.name = name
        self.author = author

schema = fields.Namespace("http://schema.org/")

class BookSchema(JsonLDSchema):
    _id = fields.Id()
    name = fields.String(schema.name)
    author = fields.String(schema.author)

    class Meta:
        rdf_type = schema.Book
        model = Book

data = {
    "@id": "http://example.com/books/1",
    "@type": "http://schema.org/Book",
    "http://schema.org/name": "Harry Potter & The Chamber of Secrets",
    "http://schema.org/author": "J. K. Rowling",
    "http://schema.org/publishedYear": 1998,
}

verified_data = BookSchema().validate_properties(data, "tests/fixtures/book_ontology.
↪owl", return_valid_data=True)

book_verified = BookSchema().load(verified_data)
#<Book(_id="http://example.com/books/1", name="Harry Potter & The Chamber of Secrets",
↪ author="J. K. Rowling")>
```

The function validate_properties has 3 arguments: `data`, `ontology` and `return_valid_data`.

`data` can be a Json-LD, a python object of the schema's model class, or a list of either of those.

`ontology` is a string pointing to the OWL ontology's location (path or URI).

`return_valid_data` is an optional argument with the default value `False`. Default behavior is to return dictionary with valid and invalid properties. Setting this to True returns the JSON-LD with only validated properties.

2.5 Annotations

Classes can also be annotated directly with schema information, removing the need to have a separate schema. This can be done by setting the `metaclass` of the model to `JsonLDAnnotation`.

```
import datetime.datetime as dt

from calamus.schema import JsonLDAnnotation
import calamus.fields as fields

schema = fields.Namespace("http://schema.org/")

class User(metaclass=JsonLDAnnotation):
    _id = fields.Id()
    birth_date = fields.Date(schema.birthDate, default=dt.now)
    name = fields.String(schema.name, default=lambda: "John")

    class Meta:
        rdf_type = schema.Person

user = User()

# dumping
User.schema().dump(user)
# or
user.dump()

# loading
u = User.schema().load({"_id": "http://example.com/user/1", "name": "Bill", "birth_
↪date": "1970-01-01 00:00"})
```

CHAPTER

THREE

SUPPORT

You can reach us on our Gitter Channel.

3.1 calamus API

3.1.1 Schema

Marshmallow schema implementation that supports JSON-LD.

class calamus.schema.**JsonLDAnnotation**(*name*, *bases*, *namespace*, ***kwargs*)

Meta-class allowing automated generation of calamus schema based on annotations.

Example:

```
import datetime.datetime as dt

from calamus import JsonLDAnnotation
import calamus.fields as fields

schema = fields.Namespace("http://schema.org/")

class User(metaclass=JsonLDAnnotation):
    class Meta:
        rdf_type = schema.Person
    _id = fields.Id()
    birth_date = fields.Date(schema.birthDate, load_default=dt.now)
    name = fields.String(schema.name, load_default=lambda: "John")

 user = User()

 # dumping
 User.schema().dump(user)
 # or
 user.dump()

 # loading
 u = User.schema().load({"_id": "http://example.com/user/1", "name": "Bill",
↪"birth_date": "1970-01-01 00:00"})
```

class calamus.schema.**JsonLDSchema**(**args*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*, *flattened=False*, *lazy=False*, *_all_objects=None*, *_visited=None*, *_top_level=True*)

Schema for a JsonLD class.

Parameters

- **flattened** (*bool*) – If the JSON-LD should be loaded/dumped in flattened form
- **lazy** (*bool*) – Enables lazy loading of nested attributes

Example:

```
from calamus import JsonLDSchema
import calamus.fields as fields
from mymodels import User
schema = fields.Namespace("http://schema.org/")
class UserSchema(JsonLDSchema):
    class Meta:
        rdf_type = schema.Person
        model = User
    _id = fields.Id()
    birth_date = fields.Date(schema.birthDate)
    name = fields.String(schema.name)
```

OPTIONS_CLASS
alias of *calamus.schema.JsonLDSchemaOpts*

get_reverse_links(*data: Mapping[str, Any]*, *field_name: str*)
Get all objects pointing to the object in data with the field field_name.

Used for unflattening a list.

make_instance(*data*, ***kwargs*)
Transform loaded dict into corresponding object.

validate_properties(*data*, *ontology*, *return_valid_data=False*, *strict=False*)
Validate JSON-LD against an ontology.

Parameters

- **data** (*Union[object, dict, list]*) – JSON-LD data or model (or list of them).
- **ontology** (*str*) – Path/URI to an ontology file.
- **return_valid_data** (*bool*) – Whether to delete invalid properties to return only valid data or else returns a dict containing valid and invalid properties, Default: `False`

class calamus.schema.**JsonLDSchemaMeta**(*name*, *bases*, *attrs*)
Meta-class for a for a JsonLDSchema class.

class calamus.schema.**JsonLDSchemaOpts**(*meta*, **args*, ***kwargs*)
Options class for *JsonLDSchema*.

Adds the following options:

- `rdf_type`: The RDF type(s) for this schema.
- `model`: The python type this schema (de-)serializes.
- `add_value_types`: Whether to add `@type` information to scalar field values.
- **id_generation_strategy: A callable(dict, obj) that generates an Id on the fly if none is set.**
  With dict being the deserialized Json-LD dict and obj being the original object.

calamus.schema.**blank_node_id_strategy**(*ret*, *obj*)
`id_generation_strategy` that creates random blank node ids.

3.1.2 Fields

Marshmallow fields for use with JSON-LD.

class calamus.fields.**AwareDateTime**(**args*, ***kwargs*)
A naive date/time field.

class calamus.fields.**BlankNodeId**(**args*, ***kwargs*)
A blank/anonymous node identifier.

property data_key
Return the (expanded) JsonLD field name.

class calamus.fields.**Boolean**(**args*, ***kwargs*)
A Boolean field.

class calamus.fields.**Date**(**args*, ***kwargs*)
A naive date/time field.

class calamus.fields.**DateTime**(**args*, *extra_formats=('%Y-%m-%d')*, ***kwargs*)
A date/time field.

class calamus.fields.**Dict**(*field_name=None*, **args*, ***kwargs*)
A dict field.

class calamus.fields.**Float**(**args*, ***kwargs*)
A float field.

class calamus.fields.**IRI**(**args*, ***kwargs*)
An external IRI reference.

class calamus.fields.**IRIReference**(*namespace*, *name*)
Represent an IRI in a namespace.

Parameters

- **namespace** (Namespace) – The Namespace this IRI is part of.
- **name** (*str*) – the property name of this IRI.

class calamus.fields.**Id**(**args*, ***kwargs*)
A node identifier.

class calamus.fields.**Integer**(**args*, ***kwargs*)
An integer field.

class calamus.fields.**List**(**args*, ***kwargs*)
A potentially ordered list using the `@list` keyword.

Parameters **ordered** (*bool*) – Whether this is an ordered (via `@list` keyword) list.

Warning: The JSON-LD flattening algorithm does not combine `@list` entries when merging nodes. So if you use `ordered=True` and flatten the output, and you have the node containing the list in multiple places in the graph, the node will get merged but its lists wont get merged (you get a list of lists instead), which means that the output can't be deserialized back to python objects.

property opts
Return parent's opts.

class calamus.fields.**NaiveDateTime**(**args*, ***kwargs*)
A naive date/time field.

class calamus.fields.**Namespace**(*namespace*, *ontology=None*)
Represents a namespace/ontology.

Parameters

- **namespace** (*str*) – The base namespace URI for this namespace.
- **ontology** (*str*) – Path to an ontology(OWL) file for this namespace.

class calamus.fields.**Nested**(**args*, ***kwargs*)

A reference to one or more nested classes.

load_single_entry(*value*, *partial*)

Loads a single nested entry from its schema.

property schema

The nested Schema object.

This method was copied from marshmallow and modified to support multiple different nested schemes.

class calamus.fields.**Raw**(*field_name=None*, **args*, ***kwargs*)

A raw field.

class calamus.fields.**RawJsonLD**(*field_name=None*, **args*, ***kwargs*)

A raw JSON-LD field.

class calamus.fields.**String**(**args*, ***kwargs*)

A string field.

class calamus.fields.**Time**(**args*, ***kwargs*)

A naive date/time field.

3.1.3 Utilities

Calamus utilities.

calamus.utils.**normalize_id**(*id_object: Union[Mapping[str, Any], Iterable[Mapping[str, Any]], str]*)

Turns a JsonLD id reference into normalized form (list of strings).

calamus.utils.**normalize_type**(*type_data*)

Normalizes a JsonLD type reference as list of string.

calamus.utils.**normalize_value**(*value*)

Normalizes a JsonLD value object to a simple value.

calamus.utils.**validate_field_properties**(*data*, *ontology*, *query=None*, *mem={'invalid': {}, 'valid': {}}*)

Validates if the field properties for data are present in the OWL ontology graph or not.

Parameters

- **data** (*dict*) – The data to validate.
- **ontology** (rdflib.Graph) – The OWL ontology graph to validate against.
- **query** (rdflib.plugins.sparql.sparql.prepareQuery) – Optional prepared query (for performance reasons).
- **mem** (*dict*) – memoization for repeated calls.

Returns Key valid has all valid properties (excluding @id and @type), invalid has all invalid properties

Return type dict

3.2 License


```
Copyright 2017-2020 - Swiss Data Science Center (SDSC)
A partnership between École Polytechnique Fédérale de Lausanne (EPFL) and
Eidgenössische Technische Hochschule Zürich (ETHZ).

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```


3.3 Contributing

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

3.3.1 Types of Contributions

Report issues / contacting developers

Report bugs on our issue tracker.

If you want to submit a bug, improvement or feature suggestions feel free to open a corresponding issue on GitHub.

If you are reporting a bug, please help us to speed up the diagnosing a problem by providing us with as much as information as possible. Ideally, that would include a step by step process on how to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for proposal discussions or epics and feel free to express your proposal on the topic. Once topic has been flushed out and we have decided how feature should be implemented, we can start implementing them.

Improvement requests

If you see room for improvement, please open an issue with a suggestion. Please motivate your suggestion by illustrating a problem it solves.

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/SwissDataScienceCenter/calamus/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 contributions are welcome :)

3.3.2 Get Started!

Ready to contribute? Here's how to set up *calamus* for local development.

1. Fork the *SwissDataScienceCenter/calamus* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/calamus.git
```

3. Ensure you have your development environment set up. For this we encourage usage of *poetry*:

```
$ poetry install
$ poetry shell
```

4. Create a branch for local development:

```
$ git checkout -b <issue_number>_<short_description>
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

```
$ poetry run pytest
```

   Before you submit a pull request, please reformat the code using black.

```
$ black
```

   You may want to set up black styling as a pre-commit hook to do this automatically:

```
$ poetry run pre-commit install
```

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

```
$ git add .
$ git commit -s
    -m "type(scope): title without verbs"
    -m "* NEW Adds your new feature."
    -m "* FIX Fixes an existing issue."
    -m "* BETTER Improves and existing feature."
    -m "* Changes something that should not be visible in release notes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.3.3 Commit message guidelines

This project is using conventional commits style for generation of changelog upon each release. Therefore, it's important that our commit messages convey what they do correctly. Commit message should always follow this pattern:

$ %{type}: %{description}

Type's used for describing commit's which will end up in changelog are `fix:` & `feat:`.

Please note that the `fix` type here is only for user-facing bug fixes and not fixes on tests or CI. For those, please use: `ci:` or `test:`

Full list of types which are in use:

- `feat:` - Used for new user-facing features.
- `fix:` - Used for fixing user-facing bugs.
- `chore:` - Used for changes which are not user-facing.
- `tests:` - Used for fixing existing or adding new tests.
- `docs:` - Used for adding more documentation.
- `refactor` - Used for changing the code structure.

3.3.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

- Make sure you agree with the license and follow the legal matter.
- The pull request should include tests and must not decrease test coverage.
- If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a doc string.
- The pull request should work for Python 3.6, 3.7 and 3.8. Check GitHub action builds and make sure that the tests pass for all supported Python versions.

3.4 Changes

3.4.1 0.4.2 (2023-02-28)

Fixes

- Fix IRI serialization when `add_value_types` is `True`.

3.4.2 0.4.1 (2022-09-01)

Fixes

- Remove usage of fields deprecated by marshmallow and don't forward calamus specific fields to prevent warnings.

3.4.3 0.4.0 (2022-04-05)

Features

- Add support for marshmallow Meta properties on annotated classes (such as `exclude`).
- Allow controlling `rdf_type` inheritance on annotated classes using the `inherit_parent_types` Boolean flag.
- Support defining marshmallow hooks on annotated classes.

3.4.4 0.3.14 (2022-03-23)

Fixes

- Fix schema inheritance not following class inheritance when using JsonLDAnnotation.

3.4.5 0.3.13 (2021-10-13)

Fixes

- Fix imports to reduce import time of library

3.4.6 0.3.12 (2021-09-14)

Fixes

- Remove SPARQL from rdflib extras to fix source distribution builds

3.4.7 0.3.11 (2021-09-14)

Fixes

- Update rdflib to 6.0.0 to fix compatibility issues with rdflib-jsonld

3.4.8 0.3.10 (2021-08-24)

Fixes

- Fix deserialization of @reverse fields with no data (PR #63)

3.4.9 0.3.9 (2021-08-23)

Fixes

- Fixes empty `IRI` serialization and `@reverse` serialization of `IRI`. (PR #61)

3.4.10 0.3.8 (2021-04-26)

Features

- Added a new field type RawJsonLD, that properly handles deserialization of raw field containing JSON-LD. (PR #58)

3.4.11 0.3.7 (2021-02-24)

Features

- Upgraded RDFLib to version 5.0.0. (PR #56)

3.4.12 0.3.6 (2020-12-11)

Fixes

- Fixed an issue when constructing classes with default parameters that are not positional-only. (PR #53)

3.4.13 0.3.5 (2020-12-10)

Features

- Annotation support (PR #17)
- Added additional DateTime fields (PR #41)
- Added ontology verification to schema and namespace (PR #42)

Fixes

- Fixed an issue with instantiating classes that have parameters with default value in their constructor. (PR #51)

3.4.14 0.3.4 (2020-09-22)

Fixes

- Fixed lazy-loading when using flattened JSON-LD (PR #45)

3.4.15 0.3.3 (2020-09-09)

Features

- Added automated `@id` generation for on schema objects, allowing ids to be generated on the fly on serialization. (PR #23)

Fixes

- Fixed serialization of lazy-loading proxy-objects back to JSON-LD. (PR #30)
- Fixed `BlankNodeField` to be a normal field like the `Id` field. (PR #23)

3.4.16 0.3.2 (2020-08-07)

Fixes

- Fixed an issue where deserializing models with keyword arguments in their constructor raises an exception. (PR #39)

3.4.17 0.3.1 (2020-08-03)

Features

- Added a `Dict` field that simply passes along the contained dictionary (which should be valid JSON-LD already). Added a `Raw` field that just returns the contained value. Added support for `add_value_types` at the field level. (PR #31)

Fixes

- Fixed an issue with sorting of schemes inside `Nested` fields. Original sort order is now preserved to allow users to specify precedence (in case of ambiguous types). Fixed the propagation of the `flattened` keyword to child schemes. Fixed `List` field deserialization calling the wrong `super()` method. (PR #31)

3.4.18 0.3.0 (2020-06-30)

Features

- Added lazy loading support (#12)

3.4.19 0.2.0 (2020-05-08)

Features

- Added IRIField (#24)
- Added BooleanField (1a93bdd)
- Added `init_name` setting to fields for cases where the name of the property differs from the name in the `__init__` method

Fixes

- Fixed an issue where `fields.Nested` would not work when used inside `fields.List`

3.4.20 0.1.2 (2020-05-08)

Features

- Allow serializing to a flat list (#5) (4289d86)
- Allow deserializing from a flat list (#4) (e8d56b3)

3.4.21 0.1.1 (2020-05-01)

Features

- Initial public release of calamus.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

calamus.fields, 11
calamus.schema, 9
calamus.utils, 12

INDEX

A

AwareDateTime (*class in calamus.fields*), 11

B

blank_node_id_strategy() (*in module calamus.schema*), 10
BlankNodeId (*class in calamus.fields*), 11
Boolean (*class in calamus.fields*), 11

C

calamus.fields
    module, 11
calamus.schema
    module, 9
calamus.utils
    module, 12

D

data_key() (*calamus.fields.BlankNodeId property*), 11
Date (*class in calamus.fields*), 11
DateTime (*class in calamus.fields*), 11
Dict (*class in calamus.fields*), 11

F

Float (*class in calamus.fields*), 11

G

get_reverse_links() (*calamus.schema.JsonLDSchema method*), 10

I

Id (*class in calamus.fields*), 11
Integer (*class in calamus.fields*), 11
IRI (*class in calamus.fields*), 11
IRIReference (*class in calamus.fields*), 11

J

JsonLDAnnotation (*class in calamus.schema*), 9
JsonLDSchema (*class in calamus.schema*), 9
JsonLDSchemaMeta (*class in calamus.schema*), 10
JsonLDSchemaOpts (*class in calamus.schema*), 10

L

List (*class in calamus.fields*), 11
load_single_entry() (*calamus.fields.Nested method*), 12

M

make_instance() (*calamus.schema.JsonLDSchema method*), 10
module
    calamus.fields, 11
    calamus.schema, 9
    calamus.utils, 12

N

NaiveDateTime (*class in calamus.fields*), 11
Namespace (*class in calamus.fields*), 11
Nested (*class in calamus.fields*), 12
normalize_id() (*in module calamus.utils*), 12
normalize_type() (*in module calamus.utils*), 12
normalize_value() (*in module calamus.utils*), 12

O

OPTIONS_CLASS (*calamus.schema.JsonLDSchema attribute*), 10
opts() (*calamus.fields.List property*), 11

R

Raw (*class in calamus.fields*), 12
RawJsonLD (*class in calamus.fields*), 12

S

schema() (*calamus.fields.Nested property*), 12
String (*class in calamus.fields*), 12

T

Time (*class in calamus.fields*), 12

V

validate_field_properties() (*in module calamus.utils*), 12

`validate_properties()` (*calamus.schema.JsonLDSchema method*), 10