uMongo Documentation

Release 0.6.1

Emmanuel Leblond

April 18, 2016

Contents

μMongo is a Python MongoDB ODM. It inception comes from two needs: the lack of async ODM and the difficulty to do document (un)serialization with existing ODMs.

From this point, μMongo made a few design choices:

- Stay close to the standards MongoDB driver to keep the same API when possible: use `find({"field": "value"})` like usual but retrieve your data nicely OO wrapped !
- Work with multiple drivers (PyMongo, TxMongo, motor_asyncio and mongomock for the moment)
- Tight integration with Marshmallow serialization library to easily dump and load your data with the outside world
- Free software: MIT license
- Test with 90%+ coverage ;-)

Quick example

```
from datetime import datetime
from pymongo import MongoClient
from umongo import Document, fields, validate

db = MongoClient().test

class User(Document):
    email = fields.EmailField(required=True, unique=True)
    birthday = fields.DateTimeField(validate=validate.Range(min=datetime(1900, 1, 1)))
    friends = fields.ListField(fields.ReferenceField("User"))

    class Meta:
        collection = db.user

goku = User(email='goku@sayen.com', birthday=datetime(1984, 11, 20))
goku.commit()
vegeta = User(email='vegeta@over9000.com', friends=[goku])
vegeta.commit()

vegeta.friends
# [ObjectId('570ddb311d41c89cabceeddb')]
vegeta.dump()
# {id': '570ddb311d41c89cabceeddc', 'email': 'vegeta@over9000.com', friends': ['570ddb2a1d41c89cabcee

User.find_one({"email": 'goku@sayen.com'})
# <object Document __main__.User({'_id': ObjectId('570ddb2a1d41c89cabceeddb'), 'friends': <object umo
#                                 'email': 'goku@sayen.com', 'birthday': datetime.datetime(1984, 11,
```

Get it now:

```
$ pip install umongo
```

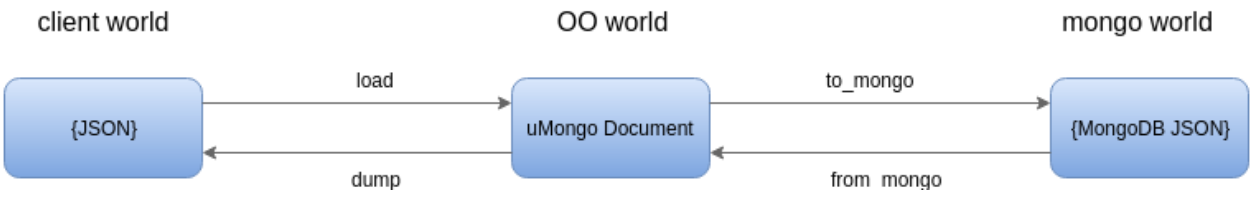
CHAPTER 1

Contents:

1.1 Tutorial

1.1.1 Base concepts

In μMongo 3 worlds are considered:



Client world

This is the data from outside μMongo, it can be JSON dict from your web framework (i.e. `request.get_json()` with flask or `json.loads(request.raw_post_data)` in django) or it could be regular Python dict with Python-typed data

JSON dict example

```
>>> {"str_field": "hello world", "int_field": 42, "date_field": "2015-01-01T00:00:00Z"}
```

Python dict example

```
>>> {"str_field": "hello world", "int_field": 42, "date_field": datetime(2015, 1, 1)}
```

To be integrated into μMongo, those data need to be unserialized. Same thing to leave μMongo they need to be serialized (under the hood μMongo uses marshmallow schema). The unserialization operation is done automatically when instantiating a *umongo.Document*. The serialization is done when calling *umongo.Document.dump()* on a document instance.

Object Oriented world

So what's good about *umongo.Document* ? Well it allows you to work with your data as Objects and to guarantee there validity against a model.

First let's define a document with few *umongo.fields*

```
class Dog(Document):
    name = fields.StrField(required=True)
    breed = fields.StrField(missing="Mongrel")
    birthday = fields.DateTimeField()
```

Note that each field can be customized with special attributes like `required` (which is pretty self-explanatory) or `missing` (if the field is missing during unserialization it will take this value).

Now we can play back and forth between OO and client worlds

```
>>> client_data = {'name': 'Odwin', 'birthday': '2001-09-22T00:00:00Z'}
>>> odwin = Dog(**client_data)
>>> odwin.breed
"Mongrel"
>>> odwin.birthday
datetime.datetime(2001, 9, 22, 0, 0)
>>> odwin.breed = "Labrador"
>>> odwin.dump()
{'birthday': '2001-09-22T00:00:00+00:00', 'breed': 'Labrador', 'name': 'Odwin'}
```

Note: You can access the data as attribute (i.e. `odwin.name`) or as item (i.e. `odwin['name']`). The latter is specially useful if one of your field name clash with *umongo.Document*'s attributes.

OO world enforces model validation for each modification

```
>>> odwin.bad_field = 42
[...]
AttributeError: bad_field
>>> odwin.birthday = "not_a_date"
[...]
ValidationError: Not a valid datetime.
```

Object orientation means inheritance, of course you can do that

```
class Animal(Document):
    breed = fields.StrField()
    birthday = fields.DateTimeField()

    class Meta:
        allow_inheritance = True
        abstract = True

class Dog(Animal):
    name = fields.StrField(required=True)

class Duck(Animal):
    pass
```

Note the `Meta` subclass, it is used (along with inherited Meta classes from parent documents) to configure the document class, you can access this final config through the `opts` attribute.

Here we use this to allow `Animal` to be inheritable and to make it abstract.

```
>>> Animal.opts
<DocumentOpts(abstract=True, allow_inheritance=True, is_child=False, base_schema_cls=<class 'umongo.s
>>> Dog.opts
<DocumentOpts(abstract=False, allow_inheritance=False, is_child=False, base_schema_cls=<class 'umongo
>>> class NotAllowedSubDog(Dog): pass
```

```
[...]
DocumentDefinitionError: Document <class '__main__.Dog'> doesn't allow inheritance
>>> Animal(breed="Mutant")
[...]
AbstractDocumentError: Cannot instantiate an abstract Document
```

Mongo world

What the point of a MongoDB ODM without MongoDB ? So here it is !

Mongo world consist of data returned in a format comprehensible by a MongoDB driver (pymongo for instance).

```
>>> odwin.to_mongo()
{'birthday': datetime.datetime(2001, 9, 22, 0, 0), 'name': 'Odwin'}
```

Well it our case the data haven't change much (if any !). Let's consider something more complex:

```
class Dog(Document):
    name = fields.StrField(attribute='_id')
```

Here we decided to use the name of the dog as our `_id` key, but for readability we keep it as `name` inside our document.

```
>>> odwin = Dog(name='Odwin')
>>> odwin.dump()
{'name': 'Odwin'}
>>> odwin.to_mongo()
{'_id': 'Odwin'}
>>> Dog.build_from_mongo({'_id': 'Scruffy'}).dump()
{'name': 'Scruffy'}
```

Note: If no field refers to `_id` in the document, a dump-only field `id` will be automatically added:

```
>>> class AutoId(Document):
...     pass
>>> AutoId.find_one()
<object Document __main__.AutoId({'_id': ObjectId('5714b9a61d41c8feb01222c8')})>
```

But what about if we what to retrieve the `_id` field whatever it name is ? No problem, use the `pk` property:

```
>>> odwin.pk
'Odwin'
>>> Duck().pk
None
```

Ok so now we got our data in a way we can insert it to MongoDB through our favorite driver. In fact most of the time you don't need to use `to_mongo` directly. Instead you should configure (remember the `Meta` class ?) you document class with a collection to insert into:

```
>>> db = pymongo.MongoClient().umongo_test
>>> class Dog(Document):
...     name = fields.StrField(attribute='_id')
...     breed = fields.StrField(missing="Mongrel")
...     class Meta:
...         collection = db.dog
```

Note: Often in more complex applications you won't have your driver ready when defining your documents. In such case you should use instead `lazy_collection` with a lazy loader depending of your driver:

```
def get_collection():
    return txmongo.MongoConnection()['lazy_db_doc']

class LazyDBDoc(Document):
    class Meta:
        lazy_collection = txmongo_lazy_loader(get_collection)
```

This way you will be able to `commit` your changes into the database:

```
>>> odwin = Dog(name='Odwin', breed='Labrador')
>>> odwin.commit()
```

You get also access to Object Oriented version of your driver methods:

```
>>> Dog.find()
<umongo.dal.pymongo.WrappedCursor object at 0x7f169851ba68>
>>> next(Dog.find())
<object Document __main__.Dog({'_id': 'Odwin', 'breed': 'Labrador'})>
Dog.find_one({'_id': 'Odwin'})
<object Document __main__.Dog({'_id': 'Odwin', 'breed': 'Labrador'})>
```

1.1.2 Multi-driver support

For the moment all examples has been done with pymongo, but thing are pretty the same with other drivers, just change the `collection` and you're good to go:

```
>>> db = motor.motor_asyncio.AsyncIOMotorClient()['umongo_test']
>>> class Dog(Document):
...     name = fields.StrField(attribute='_id')
...     breed = fields.StrField(missing="Mongrel")
...     class Meta:
...         collection = db.dog
```

Of course the way you'll be calling methods will differ:

```
>>> odwin = Dog(name='Odwin', breed='Labrador')
>>> yield from odwin.commit()
>>> dogs = yield from Dog.find()
```

Note: Be careful not to mix documents with different collection type defined or unexpected thing could happened (and furthermore there is no practical reason to do that !)

1.1.3 Going further

Inheritance

Inheritance inside the same collection is achieve by adding a `_cls` field (accessible in the document as `cls`) in the document stored in MongoDB

```
>>> class Parent(Document):
...     unique_in_parent = fields.IntField(unique=True)
...     class Meta:
...         allow_inheritance = True
>>> class Child(Parent):
...     unique_in_child = fields.StrField(unique=True)
>>> child = Child(unique_in_parent=42, unique_in_child='forty_two')
>>> child.cls
'Child'
>>> child.dump()
{'cls': 'Child', 'unique_in_parent': 42, 'unique_in_child': 'forty_two'}
>>> Parent().dump(unique_in_parent=22)
{'unique_in_parent': 22}
>>> [x.document for x in Parent.opts.indexes]
[{'key': SON([('unique_in_parent', 1)]), 'name': 'unique_in_parent_1', 'sparse': True, 'unique': True
```

Indexes

Warning: Indexes must be first submitted to MongoDB. To do so you should call `umongo.Document.ensure_indexes()` once for each document

In fields, `unique` attribute is implicitly handled by an index:

```
>>> class WithUniqueEmail(Document):
...     email = fields.StrField(unique=True)
>>> [x.document for x in WithUniqueEmail.opts.indexes]
[{'key': SON([('email', 1)]), 'name': 'email_1', 'sparse': True, 'unique': True}]
>>> WithUniqueEmail.ensure_indexes()
>>> WithUniqueEmail().commit()
>>> WithUniqueEmail().commit()
[...]
ValidationError: {'email': 'Field value must be unique'}
```

Note: The index params also depend of the `required`, `null` field attributes

For more custom indexes, the `Meta.indexes` attribute should be used:

```
>>> class CustomIndexes(Document):
...     name = fields.StrField()
...     age = fields.Int()
...     class Meta:
...         indexes = ('#name', 'age', ('-age', 'name'))
>>> [x.document for x in CustomIndexes.opts.indexes]
[{'key': SON([('name', 'hashed')]), 'name': 'name_hashed'},
 {'key': SON([('age', 1), ]), 'name': 'age_1'},
 {'key': SON([('age', -1), ('name', 1)]), 'name': 'age_-1_name_1'}
```

Indexes can be passed as:

- a string with an optional direction prefix (i.e. `"my_field"`)
- a list of string with optional direction prefix for compound indexes (i.e. `["field1", "-field2"]`)
- a `pymongo.IndexModel` object

- a dict used to instantiate an `pymongo.IndexModel` for custom configuration (i.e. `{'key': ['field1', 'field2'], 'expireAfterSeconds': 42}`)

Allowed direction prefix are:

- `+` for ascending
- `-` for descending
- `$` for text
- `#` for hashed

Note: If no direction prefix is passed, ascending is assumed

In case of a field defined in a child document, it index is automatically compounded with the `_cls`

```
>>> class Parent(Document):
...     unique_in_parent = fields.IntField(unique=True)
...     class Meta:
...         allow_inheritance = True
>>> class Child(Parent):
...     unique_in_child = fields.StrField(unique=True)
...     class Meta:
...         indexes = ['#unique_in_parent']
>>> [x.document for x in Child.opts.indexes]
[{'name': 'unique_in_parent_1', 'sparse': True, 'unique': True, 'key': SON([('unique_in_parent', 1)])
 {'name': 'unique_in_parent_hashed__cls_1', 'key': SON([('unique_in_parent', 'hashed'), ('_cls', 1)])
 {'name': '_cls_1', 'key': SON([('_cls', 1)])},
 {'name': 'unique_in_child_1__cls_1', 'sparse': True, 'unique': True, 'key': SON([('unique_in_child',
```

1.2 API Reference

1.2.1 Document

class umongo.**Document**(***kwargs*)

classmethod build_from_mongo(*data*, *partial=False*, *use_cls=False*)
Create a document instance from MongoDB data

Parameters

- **data** – data as retrieved from MongoDB
- **use_cls** – if the data contains a `_cls` field, use it determine the Document class to instanciate

clear_modified()
Reset the list of document's modified items

dbref
Return a pymongo DBRef instance related to the document

dump(*schema=None*)
Dump the document: return a `dict`

Parameters **schema** – use this schema for the dump instead of the default one

from_mongo(*data*, *partial=False*)
Update the document with the MongoDB data

Parameters **data** – data as retrieved from MongoDB

pk
Return the document's primary key (i.e. *_id* in mongo notation) or None if not available yet

to_mongo(*update=False*)

update(*data*, *schema=None*)
Update the document with the given data

Parameters **schema** – use this schema for the load instead of the default one

class umongo.meta.**DocumentOpts**(*name*, *nmspc*, *bases*)

1.2.2 Fields

class umongo.fields.**DictField**(**args*, ***kwargs*)

class umongo.fields.**ListField**(**args*, ***kwargs*)

map_to_field(*mongo_path*, *path*, *func*)
Apply a function to every field in the schema

class umongo.fields.**StringField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**UUIDField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**NumberField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**IntegerField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**DecimalField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**BooleanField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**FormattedStringField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**FloatField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**DateTimeField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**UrlField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

umongo.fields.**URLField**
alias of *UrlField*

class umongo.fields.**EmailField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

umongo.fields.**StrField**
alias of *StringField*

umongo.fields.**BoolField**
alias of *BooleanField*

umongo.fields.**IntField**
alias of *IntegerField*

class umongo.fields.**ConstantField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)

class umongo.fields.**ObjectIdField**(**args*, *io_validate=None*, *unique=False*, ***kwargs*)
Marshmallow field for `bson.ObjectId`

class umongo.fields.**ReferenceField**(*document_cls*, **args*, *reference_cls=<class 'umongo.data_objects.Reference'>*, ***kwargs*)

document_cls

class umongo.fields.**GenericReferenceField**(**args*, *reference_cls=<class 'umongo.data_objects.Reference'>*, ***kwargs*)

class umongo.fields.**EmbeddedField**(*embedded_document_cls*, **args*, ***kwargs*)

map_to_field(*mongo_path*, *path*, *func*)
Apply a function to every field in the schema

1.2.3 Data objects

class umongo.data_objects.**EmbeddedDocument**(***kwargs*)

Schema
alias of EmbeddedDocumentSchema

clear_modified()

dump()

from_mongo(*data*)

is_modified()

schema = <EmbeddedDocumentSchema(many=False, strict=False)>

set_modified()

to_mongo(*update=False*)

class umongo.data_objects.**List**(*container_field*, **args*, ***kwargs*)

append(*obj*)

clear(**args*, ***kwargs*)

extend(*iterable*)

pop(**args*, ***kwargs*)

remove(**args*, ***kwargs*)

reverse(**args*, ***kwargs*)

sort(**args*, ***kwargs*)

class umongo.data_objects.**Reference**(*document_cls*, *pk*)

io_fetch(*no_data=False*)

1.2.4 Exceptions

exception umongo.exceptions.**AbstractDocumentError**

exception umongo.exceptions.**AlreadyRegisteredDocumentError**

exception umongo.exceptions.**DeleteError**

exception umongo.exceptions.**DocumentDefinitionError**

exception umongo.exceptions.**FieldNotLoadedError**

exception umongo.exceptions.**MissingSchemaError**

exception umongo.exceptions.**NoCollectionDefinedError**

exception umongo.exceptions.**NoCompatibleDalError**

exception umongo.exceptions.**NoDBDefinedError**

exception umongo.exceptions.**NotCreatedError**

exception umongo.exceptions.**NotRegisteredDocumentError**

exception umongo.exceptions.**UMongoError**

exception umongo.exceptions.**UpdateError**

1.3 Contributing

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

You can contribute in many ways:

1.3.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/touilleMan/umongo/issues.

If you are 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.

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 features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

uMongo could always use more documentation, whether as part of the official uMongo docs, in docstrings, 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/touilleMan/umongo/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 :)

1.3.2 Get Started!

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

1. Fork the *umongo* repo on GitHub.
2. Clone your fork locally:

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

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv umongo
$ cd umongo/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 umongo tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. 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
```

7. Submit a pull request through the GitHub website.

1.3.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/touilleMan/umongo/pull_requests and make sure that the tests pass for all supported Python versions.

1.3.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_umongo
```

1.4 Credits

1.4.1 Development Lead

- Emmanuel Leblond <emmanuel.leblond@gmail.com>

1.4.2 Contributors

None yet. Why not be the first?

1.5 History

1.5.1 0.6.1 (2016-4-13)

- Add `<dal>_lazy_loader` to configure Document's lazy_collection

1.5.2 0.6.0 (2016-4-12)

- Heavy improvements everywhere !

1.5.3 0.1.0 (2016-1-22)

- First release on PyPI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Misc

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

u

R

S

T

U