uMongo Documentation

Release 2.0.1

Scille SAS

Mar 25, 2019

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
- i18n integration to localize validation error messages
- Free software: MIT license
- Test with 90%+ coverage ;-)

Quick example

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

db = MongoClient().test
instance = Instance(db)

@instance.register
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

# Make sure that unique indexes are created
User.ensure_indexes()

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
# <object umongo.data_objects.List([<object umongo.dal.pymongo.
↪PyMongoReference(document=User, pk=ObjectId('5717568613adf27be6363f78'))>])>
vegeta.dump()
# {id': '570ddb311d41c89cabceeddc', 'email': 'vegeta@over9000.com', friends': [
↪'570ddb2a1d41c89cabceeddb']}
User.find_one({"email": 'goku@sayen.com'})
```

(continues on next page)

(continued from previous page)

```
# <object Document __main__.User({'id': ObjectId('570ddb2a1d41c89cabceeddb'), 'friends
↪': <object umongo.data_objects.List([])>,
#                                 'email': 'goku@sayen.com', 'birthday': datetime.
↪datetime(1984, 11, 20, 0, 0)})>
```

Get it now:

```
$ pip install umongo           # This installs umongo with pymongo
$ pip install my-mongo-driver  # Other MongoDB drivers must be installed manually
```

Or to get it along with the MongoDB driver you're planing to use:

```
$ pip install umongo[motor]
$ pip install umongo[txmongo]
$ pip install umongo[mongomock]
```

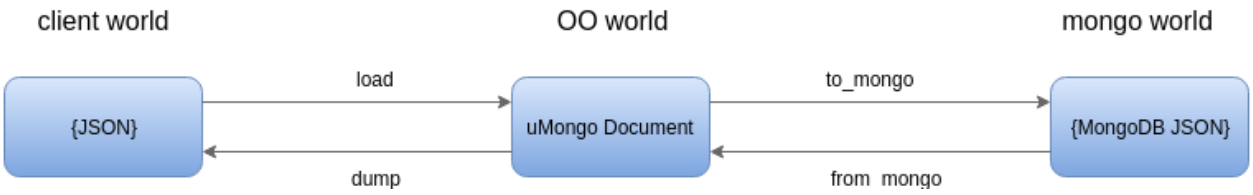
CHAPTER 1

Contents:

1.1 User guide

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*

```
@instance.register
class Dog(Document):
    name = fields.StrField(required=True)
    breed = fields.StrField(default="Mongrel")
    birthday = fields.DateTimeField()
```

First don't pay attention to the `@instance.register`, this is for later ;-)

Note that each field can be customized with special attributes like `required` (which is pretty self-explanatory) or `default` (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 clashes 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

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

    class Meta:
        allow_inheritance = True
        abstract = True
```

(continues on next page)

(continued from previous page)

```
@instance.register
class Dog(Animal):
    name = fields.StrField(required=True)

@instance.register
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(instance=<umongo.frameworks.PyMongoInstance object at 0x7efe7daa9320>,␣
↪template=<Document template class '__main__.Animal'>, abstract=True, allow_
↪inheritance=True, collection_name=None, is_child=False, base_schema_cls=<class
↪'umongo.schema.Schema'>, indexes=[], offspring={<Implementation class '__main__.Duck
↪'>, <Implementation class '__main__.Dog'>})>
>>> Dog.opts
<DocumentOpts(instance=<umongo.frameworks.PyMongoInstance object at 0x7efe7daa9320>,␣
↪template=<Document template class '__main__.Dog'>, abstract=False, allow_
↪inheritance=False, collection_name=dog, is_child=False, base_schema_cls=<class
↪'umongo.schema.Schema'>, indexes=[], offspring=set())>
>>> 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:

```
@instance.register
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 can directly ask the document to `commit` it changes in 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'})>
```

You can also access the collection used by the document at any time (for example to do more low-level operations):

```
>>> Dog.collection
Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_
↪aware=False, connect=True), 'test'), 'dog')
```

Note: By default the collection to use is the snake-cased version of the document's name (e.g. `Dog` => `dog`, `HTTPError` => `http_error`). However, you can configure (remember the `Meta` class ?) the collection to use for a document with the `collection_name` meta attribute.

1.1.2 Multi-driver support

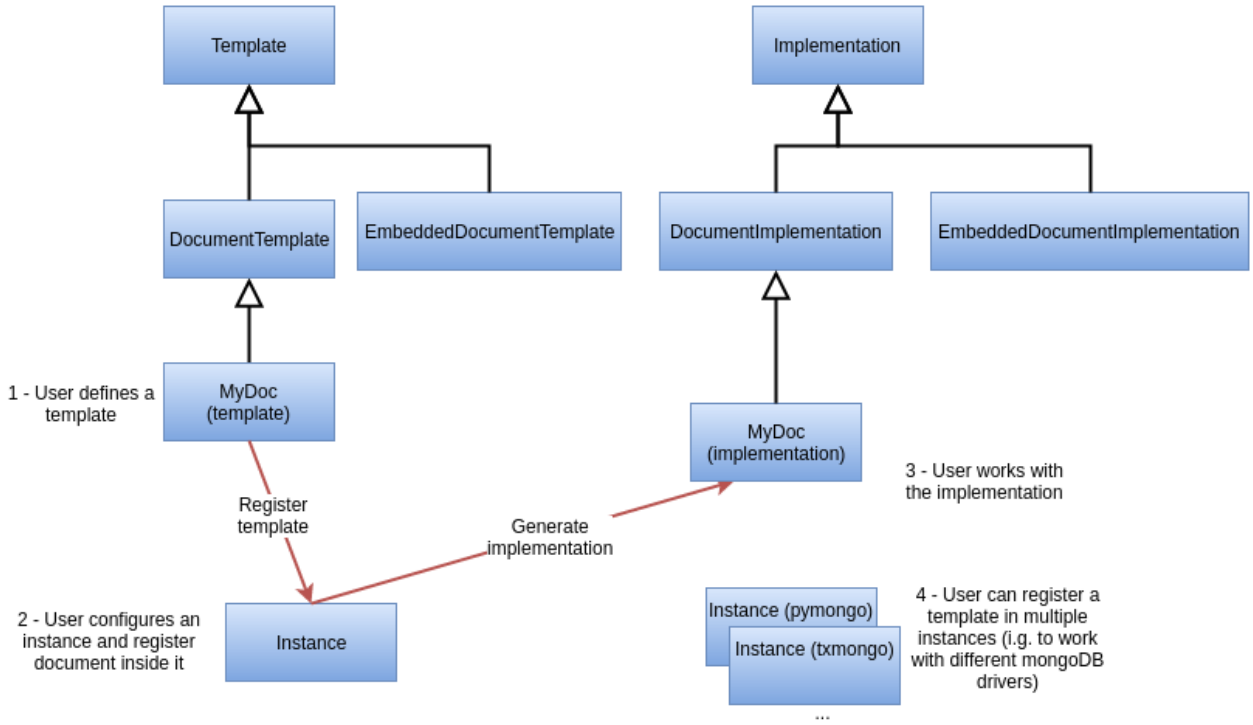
Remember the `@insance.register` ? That's now it kicks in !

The idea behind μMongo is to allow the same document definition to be used with diferent mongoDB drivers.

To achieve that the user only define document templates. Templates which will be implemented when registered by an instance:

Basically an instance provide three informations:

- the mongoDB driver type to use
- the database to use



- the documents implemented

This way a template can be implemented by multiple instances, this can be useful for example to:

- store the same documents in differents databases
- define an instance with async driver for a web server and a sync one for shell interactions

But enough of theory, let's create our first instance !

```
>>> from umongo import Instance
>>> import pymongo
>>> con = pymongo.MongoClient()
>>> instance1 = Instance(con.db1)
>>> instance2 = Instance(con.db2)
```

Now we can define & register documents, then work with them:

```
>>> class Dog(Document):
...     pass
>>> Dog  # mark as a template in repr
<Template class '__main__.Dog'>
>>> Dog.is_template
True
>>> DogInstance1Impl = instance1.register(Dog)
>>> DogInstance1Impl  # mark as an implementation in repr
<Implementation class '__main__.Dog'>
>>> DogInstance1Impl.is_template
False
>>> DogInstance2Impl = instance2.register(Dog)
>>> DogInstance1Impl().commit()
>>> DogInstance1Impl.count_documents()
1
```

(continues on next page)

(continued from previous page)

```
>>> DogInstance2Impl.count_documents()
0
```

Note: You can use `instance.register` as a decoration to replace the template by it implementation. This is expecially useful if you only use a single instance:

```
>>> @instance.register
... class Dog(Document):
...     pass
>>> Dog().commit()
```

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

```
>>> from umongo import TxMongoInstance
>>> instance = TxMongoInstance()
>>> @instance.register
... class Dog(Document):
...     pass
>>> # Don't try to use Dog (except for inheritance) now !
>>> db = create_txmongo_database()
>>> instance.init(db)
>>> # Now instance is ready
>>> yield Dog().commit()
```

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

```
>>> db = motor.motor_asyncio.AsyncIOMotorClient()['umongo_test']
>>> instance = Instance(db)
>>> @instance.register
... class Dog(Document):
...     name = fields.StrField(attribute='_id')
...     breed = fields.StrField(default="Mongrel")
```

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()
```

1.1.3 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

```
>>> @instance.register
... class Parent(Document):
...     unique_in_parent = fields.IntField(unique=True)
...     class Meta:
```

(continues on next page)

(continued from previous page)

```
...         allow_inheritance = True
>>> @instance.register
... 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(unique_in_parent=22).dump()
{'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}]
```

Warning: You must `register` a parent before it child inside a given instance.

1.1.4 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:

```
>>> @instance.register
... 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:

```
>>> @instance.register
... 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'}
```

Note: `Meta.indexes` should use the names of the fields as they appear in database (i.e. given a field `nick = StrField(attribute='nk')`, you refer to it in `Meta.indexes` as `nk`)

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`

```
>>> @instance.register
... class Parent(Document):
...     unique_in_parent = fields.IntField(unique=True)
...     class Meta:
...         allow_inheritance = True
>>> @instance.register
... 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), ('_cls', 1)])}]
```

1.1.5 I18n

μMongo provides a simple way to work with i18n (internationalization) through the `umongo.set_gettext()`, for example to use python's default gettext:

```
from umongo import set_gettext
from gettext import gettext
set_gettext(gettext)
```

This way each error message will be passed to the custom `gettext` function in order for it to return the localized version of it.

See examples/flask for a working example of i18n with flask-babel.

Note: To set up i18n inside your app, you should start with messages.pot which is a translation template of all the messages used in umongo (and it dependancy marshmallow).

1.1.6 Marshmallow integration

Under the hood, μMongo heavily uses marshmallow for all it data validation work.

However an ODM has some special needs (i.e. handling `required` fields through MongoDB's unique indexes) that force to extend marshmallow base types.

In short, you should not try to use marshmallow base types (`marshmallow.Schema`, `marshmallow.fields.Field` or `marshmallow.validate.Validator` for instance) in a μMongo document but instead use their μMongo equivalents (respectively *`umongo.abstract.BaseSchema`*, *`umongo.abstract.BaseField`* and *`umongo.abstract.BaseValidator`*).

Now let's go back to the *Base concepts*, the schema contains a little... simplification !

According to it, the client and OO worlds are made of the same data, but only in a different form (serialized vs object oriented). However it happened pretty often the API you want to provide doesn't strictly follow your datamodel (e.g. you don't want to display or allow modification of the passwords in your */users* route)

Let's go back to our *Dog* document, in real life you can rename your dog but not change it breed. So in our user API we should have a schema that enforce this !

```
>>> DogMaSchema = Dog.schema.as_marshmallow_schema()
```

As you can imagine, `as_marshmallow_schema` convert the original umongo's schema into a pure marshmallow schema. This way we can now customize it by subclassing it:

```
>>> class PatchDogSchema(DogMaSchema):
...     class Meta:
...         fields = ('name', )
>>> patch_dog_schema = PatchDogSchema()
>>> patch_dog_schema.load({'name': 'Scruffy', 'breed': 'Golden retriever'}).errors
{'_schema': ['Unknown field name breed.']}
>>> ret = patch_dog_schema.load({'name': 'Scruffy'})
>>> ret.errors
{}
>>> ret.data
{'name': 'Scruffy'}
```

Finally we can integrated the validated data into OO world:

```
>>> my_dog.update(ret.data)
>>> my_dog.name
'Scruffy'
```

Note: When instantiating a custom marshmallow schema, you can use'strict=True' to make the schema raise a *ValidationError* instead of returning an error dict. This allow a better integration in umongo own error handling:

```
try:
    data, _ = patch_dog_schema.load(payload)
    my_dog.update(data)
    my_dog.commit()
except (ValidationError, UMongoError) as e:
    # error handling
```

This works great when you want to add special behavior depending of the situation. For more simple usecases we could use the marshmallow pre/post precessors . For example to simply customize the dump:

```
>>> from umongo import post_dump  # same as `from marshmallow import post_dump`
>>> @instance.register
... class Dog(Document):
...     name = fields.StrField(required=True)
...     breed = fields.StrField(default="Mongrel")
...     birthday = fields.DateTimeField()
...     @post_dump
...     def customize_dump(self, data):
...         data['name'] = data['name'].capitalize()
...         data['brief'] = "Hi ! My name is %s and I'm a %s" % (data['name'], data[
↪'breed'])"
...
>>> Dog(name='scruffy').dump()
{'name': 'Scruffy', 'breed': 'Mongrel', 'brief': "Hi ! My name is Scruffy and I'm a␣
↪Mongrel"}
```

Now let's imagine we want to allow the per-breed creation of a massive number of ducks. The API would accept a really different format that our datamodel:

```
{
    'breeds': [
        {'name': 'Mandarin Duck', 'births': ['2016-08-29T00:00:00', '2016-08-
↪31T00:00:00', ...]},
        {'name': 'Mallard', 'births': ['2016-08-27T00:00:00', ...]},
        ...
    ]
}
```

Now starting from the umongo schema would not help, so we will create our schema from scratch... almost:

```
>>> MassiveBreedSchema(marshmallow.Schema):
...     name = Duck.schema.fields['breed'].as_marshmallow_field()
...     births = marshmallow.fields.List(
...         Duck.schema.fields['birthday'].as_marshmallow_field())
>>> MassiveDuckSchema(marshmallow.Schema):
...     breeds = marshmallow.fields.List(marshmallow.fields.
↪Nested(MassiveBreedSchema))
```

Note: A custom marshmallow schema *umongo.marshmallow_bonus.SchemaFromUmongo* can be used instead of regular `marshmallow.Schema` to benefit a tighter integration with umongo (unknown field checking and field with missing value actually return the `missing` singleton instead of serializing it as *None*)

This time we directly convert umongo schema's fields into there marshmallow equivalent with `as_marshmallow_field`. Now we can build our ducks easily:

```
try:
    data, _ =  MassiveDuckSchema(strict=True).load(payload)
    ducks = []
    for breed in data['breeds']:
        for birthday in breed['births']:
            duck = Duck(breed=breed['name']), birthday=birthday)
            duck.commit()
            ducks.append(duck)
except ValidationError as e:
    # Error handling
    ...
```

One final thought: field's `missing` and `default` attributes are not handled the same in marshmallow and umongo.

In marshmallow `default` contains the value to use during serialization (i.e. calling `schema.dump(doc)`) and `missing` the value for deserialization.

where the field is

In umongo however there is only a `default` attribute which will be used when creating (or loading from user world) a document where this field is missing. This is because you don't need to control how umongo will store the document in mongo world.

So when you use `as_marshmallow_field`, the resulting marshmallow field's `missing``&``default` will be by default both infered from the umongo's `default` field. You can want to overwrite this behavior by using `marshmallow_missing`/`marshmallow_default` attributes:

```
@instance.register
class Employee(Document):
    name = fields.StrField(default='John Doe')
    birthday = fields.DateTimeField(marshmallow_missing='2000-01-01T00:00:00Z')
    # You can use `missing` singleton to overwrite `default` field inference
    skill = fields.StrField(default='Dummy', marshmallow_default=missing)

ret = Employee.schema.as_marshmallow_schema()().load({})
assert ret.data == {'name': 'John Doe', 'birthday': datetime(2000, 1, 1, 0, 0,␣
↪tzinfo=tzutc()), 'skill': 'Dummy'}
ret = Employee.schema.as_marshmallow_schema()().dump({})
assert ret.data == {'name': 'John Doe', 'birthday': '2000-01-01T00:00:00+00:00'}  #␣
↪Note `skill` hasn't been serialized
```

1.1.7 Field validate & io_validate

Fields can be configured with special validators through the `validate` attribute:

```
from umongo import Document, fields, validate

@instance.register
class Employee(Document):
    name = fields.StrField(validate=[validate.Length(max=120), validate.Regexp(r"[a-
↪zA-Z ']+")])
    age = fields.IntField(validate=validate.Range(min=18, max=65))
    email = fields.StrField(validate=validate.Email())
    type = fields.StrField(validate=validate.OneOf(['private', 'sergeant', 'general
↪'']))
```

Those validators will be enforced each time a field is modified:

```
>>> john = Employee(name='John Rambo')
>>> john.age = 99  # it's not his war anymore...
[...]
ValidationError: ['Must be between 18 and 65.']
```

Now sometime you'll need for your validator to query your database (this is mainly done to validate a *umongo.data_objects.Reference*). For this need you can use the `io_validate` attribute. This attribute should get passed a function (or a list of functions) that will do database access in accordance with the used mongodb driver.

For example with Motor-asyncio driver, `io_validate`'s functions will be wrapped by `asyncio.coroutine` and called with `yield from`.

```
from motor.motor_asyncio import AsyncIOMotorClient
db = AsyncIOMotorClient().test
instance = Instance(db)

@instance.register
class TrendyActivity(Document):
    name = fields.StrField()


@instance.register
class Job(Document):

    def _is_dream_job(field, value):
        if not (yield from TrendyActivity.find_one(name=value)):
            raise ValidationError("No way I'm doing this !")

    activity = fields.StrField(io_validate=_is_dream_job)


@asyncio.coroutine
def run():
    yield from TrendyActivity(name='Pythoning').commit()
    yield from Job(activity='Pythoning').commit()
    yield from Job(activity='Javascripting...').commit()
    # raises ValidationError: {'activity': ["No way I'm doing this !"]}
```

Warning: When converting to marshmallow with *as_marshmallow_schema* and *as_marshmallow_fields*, *io_validate* attribute will not be preserved.

1.2 API Reference

1.2.1 Instance

class umongo.instance.**BaseInstance**(*templates=()*)

Base class for instance.

Instances aims at collecting and implementing `umongo.template.Template`:

```
# Doc is a template, cannot use it for the moment
class Doc(DocumentTemplate):
    pass
```

(continues on next page)

(continued from previous page)

```
instance = Instance()
# doc_cls is the instance's implementation of Doc
doc_cls = instance.register(Doc)
# Implementations are registered as attribute into the instance
instance.Doc is doc_cls
# Now we can work with the implementations
doc_cls.find()
```

Note: Instance registration is divided between *umongo.Document* and umongo.EmbeddedDocument.

db
Database used within the instance.

register(*template*, *as_attribute=True*)
Generate an umongo.template.Implementation from the given umongo.template.Template for this instance.

Parameters

- **template** – umongo.template.Template to implement
- **as_attribute** – Make the generated umongo.template.Implementation available as this instance's attribute.

Returns The umongo.template.Implementation generated

Note: This method can be used as a decorator. This is useful when you only have a single instance to work with to directly use the class you defined:

```
@instance.register
class MyEmbedded(EmbeddedDocument):
    pass

@instance.register
class MyDoc(Document):
    emb = fields.EmbeddedField(MyEmbedded)

MyDoc.find()
```

retrieve_document(*name_or_template*)
Retrieve a *umongo.document.DocumentImplementation* registered into this instance from it name or it template class (i.e. *umongo.Document*).

retrieve_embedded_document(*name_or_template*)
Retrieve a umongo.embedded_document.EmbeddedDocumentImplementation registered into this instance from it name or it template class (i.e. umongo.EmbeddedDocument).

class umongo.**Instance**(*db*, *templates=()*)
Automatically configured instance according to the type of the provided database.

class umongo.instance.**LazyLoaderInstance**(*templates=()*)
Base class for instance with database lazy loading.

Note: This class should not be used directly but instead overloaded. See *umongo.PyMongoInstance* for example.

db
Database used within the instance.

init(*db*)
Set the database to use whithin this instance.

Note: The documents registered in the instance cannot be used before this function is called.

class umongo.**PyMongoInstance**(**args*, ***kwargs*)
umongo.instance.LazyLoaderInstance implementation for pymongo

class umongo.**TxMongoInstance**(**args*, ***kwargs*)
umongo.instance.LazyLoaderInstance implementation for txmongo

class umongo.**MotorAsyncIOInstance**(**args*, ***kwargs*)
umongo.instance.LazyLoaderInstance implementation for motor-asyncio

class umongo.**MongoMockInstance**(**args*, ***kwargs*)
umongo.instance.LazyLoaderInstance implementation for mongomock

1.2.2 Document

umongo.**Document**
Shortcut to DocumentTemplate

alias of *umongo.document.DocumentTemplate*

class umongo.document.**DocumentTemplate**(**args*, ***kwargs*)
Base class to define a umongo document.

Note: Once defined, this class must be registered inside a *umongo.instance.BaseInstance* to obtain it corresponding *umongo.document.DocumentImplementation*.

Note: You can provide marshmallow tags (e.g. *marshmallow.pre_load* or *marshmallow.post_dump*) to this class that will be passed to the marshmallow schema internally used for this document.

class umongo.document.**DocumentOpts**(*instance*, *template*, *collection_name=None*, *abstract=False*, *allow_inheritance=None*, *indexes=None*, *is_child=True*, *strict=True*, *offspring=None*)
Configuration for a document.

Should be passed as a Meta class to the `Document`

```
@instance.register
class Doc(Document):
    class Meta:
        abstract = True

assert Doc.opts.abstract == True
```

attribute	configurable in Meta	description
template	no	Origine template of the Document
instance	no	Implementation's instance
abstract	yes	Document has no collection and can only be inherited
allow_inheritance	yes	Allow the document to be subclassed
collection_name	yes	Name of the collection to store the document into
is_child	no	Document inherit of a non-abstract document
strict	yes	Don't accept unknown fields from mongo (default: True)
indexes	yes	List of custom indexes
offspring	no	List of Documents inheriting this one

class umongo.document.**DocumentImplementation**(***kwargs*)

Represent a document once it has been implemented inside a *umongo.instance.BaseInstance*.

Note: This class should not be used directly, it should be inherited by concrete implementations such as `umongo.frameworks.pymongo.PyMongoDocument`

---

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.

clone()

Return a copy of this Document as a new Document instance

All fields are deep-copied except the _id field.

collection

Return the collection used by this document class

dbref

Return a pymongo DBRef instance related to the document

dump()

Dump the document.

from_mongo(*data*, *partial=False*)

Update the document with the MongoDB data

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

is_created

Return True if the document has been commited to database

is_modified()

Returns True if and only if the document was modified since last commit.

pk

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

> **Warning:** Use `is_created` field instead to test if the document has already been commited to database given `_id` field could be generated before insertion

post_delete(*ret*)
Overload this method to get a callback after document deletion. :param ret: Pymongo response sent by the database.

> **Note:** If you use an async driver, this callback can be asynchronous.

post_insert(*ret*)
Overload this method to get a callback after document insertion. :param ret: Pymongo response sent by the database.

> **Note:** If you use an async driver, this callback can be asynchronous.

post_update(*ret*)
Overload this method to get a callback after document update. :param ret: Pymongo response sent by the database.

> **Note:** If you use an async driver, this callback can be asynchronous.

pre_delete()
Overload this method to get a callback before document deletion. :return: Additional filters dict that will be used for the query to

select the document to update.

> **Note:** If you use an async driver, this callback can be asynchronous.

pre_insert()
Overload this method to get a callback before document insertion.

> **Note:** If you use an async driver, this callback can be asynchronous.

pre_update()
Overload this method to get a callback before document update. :return: Additional filters dict that will be used for the query to

select the document to update.

> **Note:** If you use an async driver, this callback can be asynchronous.

to_mongo(*update=False*)
Return the document as a dict compatible with MongoDB driver.

Parameters **update** – if True the return dict should be used as an update payload instead of containing the entire document

update(*data*)
Update the document with the given data.

1.2.3 Abstracts

class umongo.abstract.**BaseSchema**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)
All schema used in umongo should inherit from this base schema

as_marshmallow_schema(*params=None*, *base_schema_cls=<class 'marshmallow.schema.Schema'>*, *check_unknown_fields=True*, *mongo_world=False*, *meta=None*)
Return a pure-marshmallow version of this schema class.

Parameters

- **params** – Per-field dict to pass parameters to their field creation.
- **base_schema_cls** – Class the schema will inherit from (default: `marshmallow.Schema`).
- **check_unknown_fields** – Unknown fields are considered as errors (default: True).
- **mongo_world** – If True the schema will work against the mongo world instead of the OO world (default: False).
- **meta** – Optional dict with attributes for the schema's Meta class.

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

```
>>> def func(mongo_path, path, field):
...     pass
```

class umongo.abstract.**BaseField**(**args*, *io_validate=None*, *unique=False*, *instance=None*, ***kwargs*)
All fields used in umongo should inherit from this base field.

Enabled flags	resulting index
<no flags>	
allow_none	
required	
required, allow_none	
required, unique, allow_none	unique
unique	unique, sparse
unique, required	unique
unique, allow_none	unique, sparse

Note: Even with allow_none flag, the unique flag will refuse duplicated

null value (consider unsetting the field with *del* instead)

as_marshmallow_field(*params=None*, *mongo_world=False*, ***kwargs*)
Return a pure-marshmallow version of this field.

Parameters

- **params** – Additional parameters passed to the marshmallow field class constructor.
- **mongo_world** – If True the field will work against the mongo world instead of the OO world (default: False)

default_error_messages = {'unique': 'Field value must be unique.', 'unique_compound':

deserialize(*value*, *attr=None*, *data=None*)
Deserialize `value`.

Raises **ValidationError** – If an invalid value is passed or if a required value is missing.

deserialize_from_mongo(*value*)

serialize(*attr*, *obj*, *accessor=None*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** (*str*) – The attibute or key to get from the object.
- **obj** (*str*) – The object to pull the key from.
- **accessor** (*callable*) – Function used to pull values from `obj`.

Raises **ValidationError** – In case of formatting problem

serialize_to_mongo(*obj*)

translate_query(*key*, *query*)

class umongo.abstract.**BaseValidator**(**args*, ***kwargs*)
All validators in umongo should inherit from this base validator.

class umongo.abstract.**BaseDataObject**
All data objects in umongo should inherit from this base data object.

classmethod build_from_mongo(*data*)

clear_modified()

dump()

from_mongo(*data*)

is_modified()

to_mongo(*update=False*)

1.2.4 Fields

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

translate_query(*key*, *query*)

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

as_marshmallow_field(*params=None*, *mongo_world=False*, ***kwargs*)
Return a pure-marshmallow version of this field.

Parameters

- **params** – Additional parameters passed to the marshmallow field class constructor.

- **mongo_world** – If True the field will work against the mongo world instead of the OO world (default: False)

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*, *instance=None*, ***kwargs*)

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

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

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

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

> **Warning:** This field is only supported on MongoDB 3.4+. Using it on older versions will result in uncaught errors.

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

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

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

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

class umongo.fields.**DateField**(**args*, *io_validate=None*, *unique=False*, *instance=None*, ***kwargs*)
This field converts a date to a datetime to store it as a BSON Date

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

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

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

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

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

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

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

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

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

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

as_marshmallow_field(*params=None*, *mongo_world=False*, ***kwargs*)
Return a pure-marshmallow version of this field.

Parameters

- **params** – Additional parameters passed to the marshmallow field class constructor.
- **mongo_world** – If True the field will work against the mongo world instead of the OO world (default: False)

document_cls
Return the instance's umongo.embedded_document.DocumentImplementation implementing the *document* attribute.

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

as_marshmallow_field(*params=None*, *mongo_world=False*, ***kwargs*)
Return a pure-marshmallow version of this field.

Parameters

- **params** – Additional parameters passed to the marshmallow field class constructor.
- **mongo_world** – If True the field will work against the mongo world instead of the OO world (default: False)

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

as_marshmallow_field(*params=None*, *mongo_world=False*, ***kwargs*)
Return a pure-marshmallow version of this field.

Parameters

- **params** – Additional parameters passed to the marshmallow field class constructor.
- **mongo_world** – If True the field will work against the mongo world instead of the OO world (default: False)

embedded_document_cls
Return the instance's umongo.embedded_document.EmbeddedDocumentImplementation implementing the *embedded_document* attribute.

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

nested

1.2.5 Data objects

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

append(*obj*)
Append object to the end of the list.

clear(**args*, ***kwargs*)
Remove all items from list.

clear_modified()

container_field

extend(*iterable*)
Extend list by appending elements from the iterable.

is_modified()

pop(**args*, ***kwargs*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(**args*, ***kwargs*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse(**args*, ***kwargs*)
Reverse *IN PLACE*.

set_modified()

sort(**args*, ***kwargs*)
Stable sort *IN PLACE*.

class umongo.data_objects.**Dict**(**args*, ***kwargs*)

clear_modified()

is_modified()

set_modified()

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

error_messages = {'not_found': 'Reference not found for document {document}.'}

fetch(*no_data=False*, *force_reload=False*)
Retrieve from the database the referenced document

Parameters

- **no_data** – if True, the caller is only interested in whether or not the document is present in database. This means the implementation may not retrieve document's data to save bandwidth.
- **force_reload** – if True, ignore any cached data and reload referenced document from database.

1.2.6 Marshmallow integration

umongo.marshmallow_bonus.**schema_validator_check_unknown_fields**(*self*, *data*, *original_data*)
Schema validator, raise ValidationError for unknown fields in a marshmallow schema.

example:

```
class MySchema(marshsmallow.Schema):
    # method's name is not important
    __check_unknown_fields = validates_schema(pass_original=True)(
        schema_validator_check_unknown_fields)

    # Define the rest of your schema
    ...
```

..note:: Unknown fields with *missing* value will be ignored

umongo.marshmallow_bonus.**schema_from_umongo_get_attribute**(*self*, *attr*, *obj*, *default*)

Overwrite default *Schema.get_attribute* method by this one to access umongo missing fields instead of returning *None*.

example:

```
class MySchema(marshsmallow.Schema):
    get_attribute = schema_from_umongo_get_attribute

    # Define the rest of your schema
    ...
```

class umongo.marshmallow_bonus.**SchemaFromUmongo**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Custom `marshmallow.Schema` subclass providing unknown fields checking and custom get_attribute for umongo documents.

get_attribute(*attr*, *obj*, *default*)

Overwrite default *Schema.get_attribute* method by this one to access umongo missing fields instead of returning *None*.

example:

```
class MySchema(marshsmallow.Schema):
    get_attribute = schema_from_umongo_get_attribute

    # Define the rest of your schema
    ...
```

class umongo.marshmallow_bonus.**StrictDateTime**(*load_as_tz_aware=False*, **args*, ***kwargs*)

Marshmallow DateTime field with extra parameter to control whether dates should be loaded as tz_aware or not

class umongo.marshmallow_bonus.**ObjectId**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

Marshmallow field for `bson.ObjectId`

class umongo.marshmallow_bonus.**Reference**(**args*, *mongo_world=False*, ***kwargs*)

Marshmallow field for *umongo.fields.ReferenceField*

class umongo.marshmallow_bonus.**GenericReference**(**args*, *mongo_world=False*, ***kwargs*)

Marshmallow field for *umongo.fields.GenericReferenceField*

1.2.7 Exceptions

exception umongo.exceptions.**AbstractDocumentError**

exception umongo.exceptions.**AlreadyRegisteredDocumentError**

exception umongo.exceptions.**BuilderNotDefinedError**

exception umongo.exceptions.**DeleteError**

exception umongo.exceptions.**DocumentDefinitionError**

exception umongo.exceptions.**FieldNotLoadedError**

exception umongo.exceptions.**MissingSchemaError**

exception umongo.exceptions.**NoCollectionDefinedError**

exception umongo.exceptions.**NoCompatibleBuilderError**

exception umongo.exceptions.**NoDBDefinedError**

exception umongo.exceptions.**NotCreatedError**

exception umongo.exceptions.**NotRegisteredDocumentError**

exception umongo.exceptions.**UMongoError**

exception umongo.exceptions.**UnknownFieldInDBError**

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
$ py.test tests
$ tox
```

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

Note: You need pytest>=2.8

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 I18n

There are additional steps to make changes involving translated strings.

1. Extract translatable strings from the code into messages.pot:

```
$ make extract_messages
```

2. Update flask example translation files:

```
$ make update_flask_example_messages
```

3. Update/fix translations
4. Compile new binary translation files:

```
$ make compile_flask_example_messages
```

1.3.4 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 3.5, 3.6 and 3.7. Check https://travis-ci.org/touilleMan/umongo/pull_requests and make sure that the tests pass for all supported Python versions.

1.4 Credits

1.4.1 Development Lead

- Emmanuel Leblond @touilleMan

1.4.2 Contributors

- Jérôme Lafréchoux @lafrech
- Imblc @imbolc
- @patlach42
- Serj Shevchenko @serjshevchenko

1.5 History

1.5.1 2.0.1 (2019-03-25)

Bug fixes:

- Fix deserialization of `EmbeddedDocument` containing fields overriding `_deserialize_from_mongo` (see #186).

1.5.2 2.0.0 (2019-03-18)

Features:

- *Backwards-incompatible*: `missing` attribute is no longer used in umongo fields, only `default` is used. `marshmallow_missing` and `marshmallow_default` attribute can be used to overwrite the value to use in the pure marshmallow field returned by `as_marshmallow_field` method (see #36 and #107).
- *Backwards-incompatible*: `as_marshmallow_field` does not pass `load_from`, `dump_to` and `attribute` to the pure marshmallow field anymore. It only passes `validate`, `required`, `allow_none`, `dump_only`, `load_only` and `error_messages`, as well as `default` and `missing` values inferred from umongo's `default`. Parameters prefixed with `marshmallow_` in the umongo field are passed to the pure marshmallow field and override their non-prefixed counterpart. (see #170)
- *Backwards-incompatible*: `DictField` and `ListField` don't default to empty `Dict`/`List`. To keep old behaviour, pass `dict`/`list` as default. (see #105)
- *Backwards-incompatible*: Serialize empty `Dict`/`List` as empty rather than missing (see #105).
- Round datetimes to millisecond precision in `DateTimeField`, `LocalDateTimeField` and `StrictDateTimeField` to keep consistency between object and database representation (see #172 and #175).
- Add `DateField` (see #178).

Bug fixes:

- Fix passing a default value to a `DictField`/`ListField` as a raw Python `dict`/`list` (see #78).
- The `default` parameter of a Field is deserialized and validated (see #174).

Other changes:

- Support Python 3.7 (see #181).
- *Backwards-incompatible*: Drop Python 3.4 support (see #176) and only use async/await coroutine style in asyncio framework (see #179).

1.5.3 1.2.0 (2019-02-08)

- Add `Schema` cache to `as_marshmallow_schema` (see #165).
- Add `DecimalField`. This field only works on MongoDB 3.4+. (see #162)

1.5.4 1.1.0 (2019-01-14)

- Fix bug when filtering by id in a Document subclass find query (see #145).
- Fix __getattr__ to allow copying and deepcopying Document and EmbeddedDocument (see #157).

- Add Document.clone() method (see #158).

1.5.5 1.0.0 (2018-11-29)

- Raise `UnknownFieldInDBError` when an unknown field is found in database and not using `BaseNonStrictDataProxy` (see #121)
- Fix (non fatal) crash in garbage collector when using `WrappedCursor` with mongomock
- Depend on pymongo 3.7+ (see #149)
- Pass `as_marshmallow_schema params` to nested schemas. Since this change, every field's `as_marshmallow_schema` method should expect unknown `**kwargs` (see #101).
- Pass params to container field in `ListField.as_marshmallow_schema` (see #150)
- Add `meta` kwarg to `as_marshmallow_schema` to pass a `dict` of attributes for the schema's `Meta` class (see #151)

1.5.6 0.15.0 (2017-08-15)

- Add *strict* option to (Embedded)DocumentOpts to allow loading of document with unknown fields from mongo (see #115)
- Fix fields serialization/deserialization when allow_none is True (see #69)
- Fix ReferenceFild assignment from another ReferenceField (see #110)
- Fix deletion of field proxied by a property (see #109)
- Fix StrictDateTime bonus field: _deserialize does not accept datetime.datetime instances (see #106)
- Add force_reload param to Reference.fetch (see #96)

1.5.7 0.14.0 (2017-03-03)

- Fix bug in mashmallow tag handling (see #90)
- Fix allow none in DataProxy.set (see #89)
- Support motor 1.1 (see #87)

1.5.8 0.13.0 (2017-01-02)

- Fix deserialization error with nested EmbeddedDocuments (see #84, #67)
- Add `abstract` and `allow_inheritance` options to EmbeddedDocument
- Remove buggy `as_marshmallow_schema`'s parameter `missing_accessor` (see #73, #74)

1.5.9 0.12.0 (2016-11-11)

- Replace `Document.opts.children` by `offspring` and fix grand child inheritance issue (see #66)
- Fix dependency since release of motor 1.0 with breaking API

1.5.10 0.11.0 (2016-11-02)

- data_objects `Dict` and `List` inherit builtins `dict` and `list`
- Document&EmbeddedDocument store fields passed during initialization as modified (see #50)
- Required field inside embedded document are handled correctly (see #61)
- Document support marshmallow's pre/post processors

1.5.11 0.10.0 (2016-09-29)

- Add pre/post update/insert/delete hooks (see #22)
- Provide Umongo to Marshmallow schema/field conversion with schema.as_marshmallow_schema() and field.as_marshmallow_field() (see #34)
- List and Dict inherit from collections's UserList and UserDict instead of builtins types (needed due to metaprogramming conflict otherwise)
- DeleteError and UpdateError returns the driver result object instead of the raw error dict (except for motor which only has raw error dict)

1.5.12 0.9.0 (2016-06-11)

- Queries can now be expressed with the document's fields name instead of the name in database
- `EmbeddedDocument` also need to be registered by and instance before use

1.5.13 0.8.1 (2016-05-19)

- Replace `Document.created` by `is_created` (see #14)

1.5.14 0.8.0 (2016-05-18)

- Heavy rewrite of the project, lost of API breakage
- Documents are now first defined as templates then implemented inside an Instance
- DALs has been replaced by frameworks implementations of Builder
- Fix `__getitem__` for Pymongo.Cursor wrapper
- Add `conditions` argument to Document.commit
- Add `count` method to txmongo

1.5.15 0.7.8 (2016-4-28)

- Fix setup.py style preventing release of version 0.7.7

1.5.16 0.7.7 (2016-4-28)

- Fix await error with Reference.fetch
- Pymongo is now only installed with extra flavours of umongo

1.5.17 0.7.6 (2016-4-28)

- Use extras_require to install driver along with umongo

1.5.18 0.7.5 (2016-4-23)

- Fixing await (Python >= 3.5) support for motor-asyncio

1.5.19 0.7.4 (2016-4-21)

- Fix missing package in setup.py

1.5.20 0.7.3 (2016-4-21)

- Fix setup.py style preventing from release

1.5.21 0.7.2 (2016-4-21)

- Fix crash when generating indexes on EmbeddedDocument

1.5.22 0.7.1 (2016-4-21)

- Fix setup.py not to install tests package
- Pass status to Beta

1.5.23 0.7.0 (2016-4-21)

- Add i18n support
- Add MongoMock support
- Documentation has been a lot extended

1.5.24 0.6.1 (2016-4-13)

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

1.5.25 0.6.0 (2016-4-12)

- Heavy improvements everywhere !

1.5.26 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.

Python Module Index

u

Index

S

T

U