uMongo Documentation

Release 3.1.0

Scille SAS

Dec 23, 2021

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 ;-)

µMongo requires MongoDB 4.2+ and Python 3.7+.

Quick example

```
import datetime as dt
from pymongo import MongoClient
from umongo import Document, fields, validate
from umongo.frameworks import PyMongoInstance

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

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

    class Meta:
        collection_name = "user"

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

goku = User(email='goku@sayen.com', birthday=dt.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'))>])>
```

(continues on next page)

(continued from previous page)

```
vegeta.dump()
# {id': '570ddb311d41c89cabceeddc', 'email': 'vegeta@over9000.com', friends': [
↪'570ddb2a1d41c89cabceeddb']}
User.find_one({"email": 'goku@sayen.com'})
# <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]
```

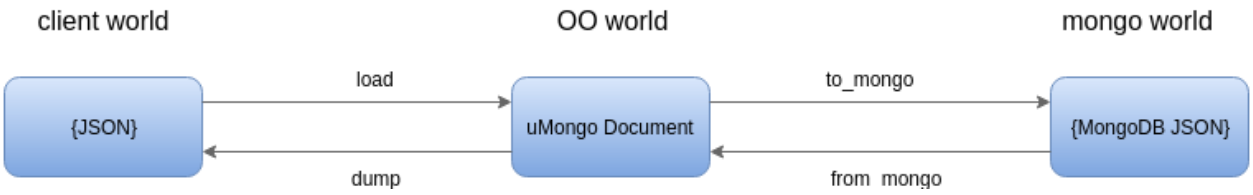
CHAPTER 1

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 a 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 a 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 deserialized and to leave μMongo they need to be serialized (under the hood μMongo uses marshmallow schema).

The deserialization 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

umongo.Document allows you to work with your data as objects and to guarantee their 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()
```

Don't pay attention to the `@instance.register` for now.

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 deserialization 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:
        abstract = True

@instance.register
class Dog(Animal):
```

(continues on next page)

(continued from previous page)

```
    name = fields.StrField(required=True)

@instance.register
class Duck(Animal):
    pass
```

The `Meta` subclass 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 inherited and to make it abstract.

```
>>> Animal.opts
<DocumentOpts(instance=<umongo.frameworks.PyMongoInstance object at 0x7efe7daa9320>,␣
↪template=<Document template class '__main__.Animal'>, abstract=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, 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

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

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

In this case, the data is unchanged. Let's consider something more complex:

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

We 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')})>
```

To retrieve the `_id` field whatever its name is, use the `pk` property:

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

Most of the time, the user doesn't need to use `to_mongo` directly. It is called internally by `umongo.Document.commit()` which is the method used to commit changes to the database.

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

μMongo provides access to Object Oriented versions of 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'})>
```

The user can also access the collection used by the document at any time to perform 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, through the `Meta` class, the collection to use for a document with the `collection_name` meta attribute.

1.1.2 Multi-driver support

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

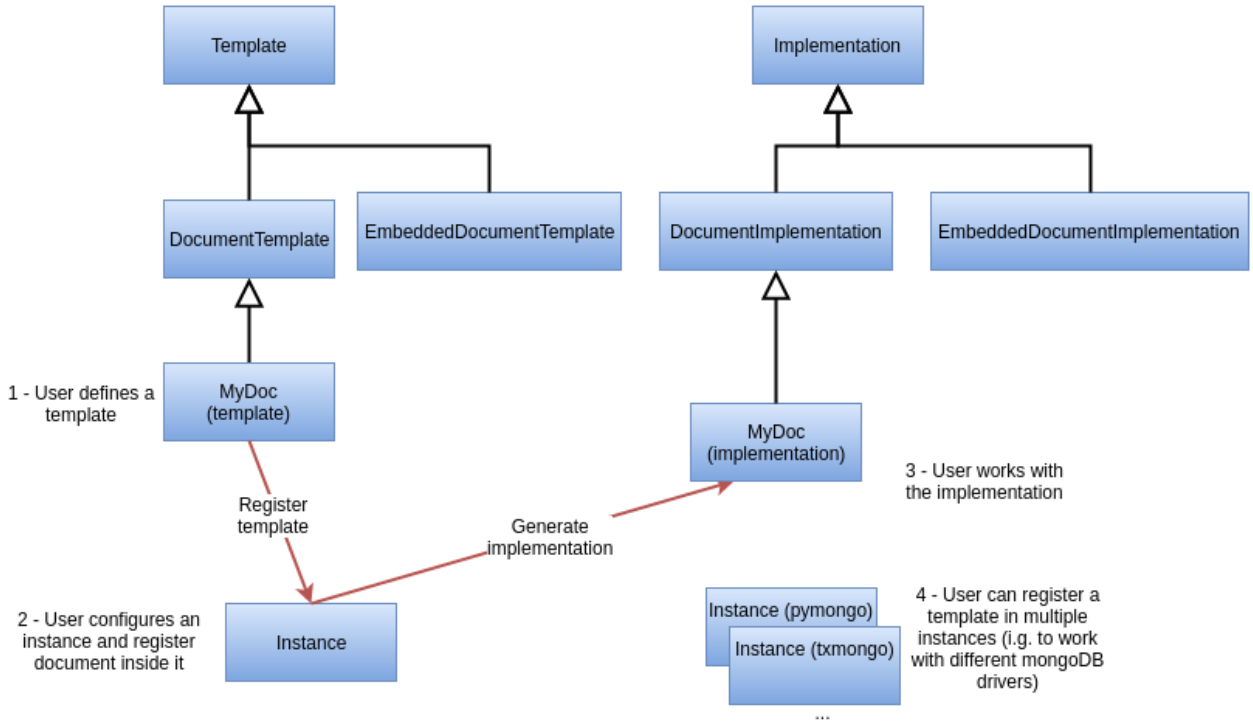
To achieve that the user only defines document templates. Templates which will be implemented when registered by an instance:

Basically an instance provide three informations:

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

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

Here's how to create and use an instance:

```
>>> from umongo.frameworks import PyMongoInstance
>>> import pymongo
>>> con = pymongo.MongoClient()
>>> instance1 = PyMongoInstance(con.db1)
>>> instance2 = PyMongoInstance(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
>>> DogInstance2Impl.count_documents()
0
```

Note: In most cases, only a single instance is used. In this case, one can use `instance.register` as a decoration

to replace the template by its implementation.

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

Note: In real-life applications, the driver connection details may not be known when registering models. For instance, when using the Flask app factory pattern, one will instantiate the instance and register model documents at import time, then pass the database connection at app init time. This can be achieved with the `set_db` method. No database interaction can be performed until a database connection is set.

```
>>> from umongo.frameworks import TxMongoInstance
>>> # Don't pass a database connection when instantiating the instance
>>> instance = TxMongoInstance()
>>> @instance.register
... class Dog(Document):
...     pass
>>> # Don't try to use Dog (except for inheritance) yet
>>> # A database connection must be set first
>>> db = create_txmongo_database()
>>> instance.set_db(db)
>>> # Now instance is ready
>>> yield Dog().commit()
```

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

```
>>> from umongo.frameworks import MotorAsyncIOInstance
>>> db = motor.motor_asyncio.AsyncIOMotorClient()['umongo_test']
>>> instance = MotorAsyncIOInstance(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)
>>> @instance.register
... class Child(Parent):
```

(continues on next page)

(continued from previous page)

```
...     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.indexes]
[{'key': SON([('unique_in_parent', 1)]), 'name': 'unique_in_parent_1', 'sparse': True,
↪ 'unique': True}]
```

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

Due to the way document instances are created from templates, fields and pre/post_dump/load methods can only be inherited from mixin classes by explicitly using a *umongo.MixinDocument*.

```
@instance.register
class TimeMixin(MixinDocument):
    date_created = fields.DateTimeField()
    date_modified = fields.DateTimeField()

@instance.register
class MyDocument(Document, TimeMixin)
    name = fields.StringField()
```

A *umongo.MixinDocument* can be inherited by both *umongo.Document* and *umongo.EmbeddedDocument* classes.

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.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.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, its index is automatically compounded with `_cls`

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

In the *Base concepts* paragraph, 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, quite often, the application API doesn't strictly exposes the datamodel (e.g. you don't want to display or allow modification of the passwords in your */users* route).

Back to our *Dog* document. In real life one can rename your dog but not change its breed. The user API should have a schema that enforces this.

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

`as_marshmallow_schema` convert the original μMongo schema into a pure marshmallow schema that can be subclassed and customized:

```
>>> 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
{'name': 'Scruffy'}
```

Finally we can integrate the validated data into OO world:

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

This works great when you want to add special behaviors 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 than 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', ...]},
        ...
    ]
}
```

Starting from the µMongo schema would not help, but one can create a new schema using pure marshmallow fields generated with the `umongo.BaseField.dump.as_marshmallow_field()` method:

```
>>> 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.schema.RemoveMissingSchema` can be used instead of regular `marshmallow.Schema` to skip missing fields when dumping a *`umongo.Document`* object.

```
try:
    data, _ = MassiveDuckSchema().load(payload)
    ducks = []
    for breed in data['breeds']:
        for birthday in breed['births']:
            duck = Duck(breed=breed['name']), birthday=birthday)
```

(continues on next page)

(continued from previous page)

```
            duck.commit()
            ducks.append(duck)
except ValidationError as e:
    # Error handling
    ...
```

Note:

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.

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

It can be useful to let all the generated marshmallow schemas inherit a custom base schema class. For instance to customize this base schema using a Meta class.

This can be done by defining a custom base schema class and passing it as a class attribute to a custom *umongo.Document* subclass.

Since the default base schema is `umongo.abstract.BaseMarshmallowSchema`, it makes sense to build from here.

```
class BaseMaSchema(umongo.abstract.BaseMarshmallowSchema):
   class Meta:
      ...  # Add custom attributes here

   # Implement custom methods here
   def custom_method(self):
      ...

 @instance.register
 class MyDocument(Document):
   MA_BASE_SCHEMA_CLS = BaseMaSchema
```

This is done at document level, but it is possible to do it in a custom base `Document` class to avoid duplication.

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.']
```

Validators may need to query the database (e.g. to validate a *umongo.data_objects.Reference*). For this need one can use the `io_validate` argument. It should be a function (or a list of functions) that will do database access in accordance with the chosen 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
from umongo.frameworks import MotorAsyncIOInstance
db = AsyncIOMotorClient().test
instance = MotorAsyncIOInstance(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 Migrating

1.2.1 Migrating from umongo 2 to umongo 3

For a full list of changes, see the CHANGELOG.

Database migration

Aside from changes in application code, migrating from umongo 2 to umongo 3 requires changes in the database.

The way the embedded documents are stored has changed. The *_cls* attribute is now only set on embedded documents that are subclasses of a concrete embedded document. Unless documents are non-strict (i.e. transparently handle unknown fields, default is strict), the database must be migrated to remove the *_cls* fields on embedded documents that are not subclasses of a concrete document.

This change is irreversible. It requires the knowledge of the application model (the document and embedded document classes).

umongo provides dedicated framework specific `Instance` subclasses to help on this.

A simple procedure to build a migration tool is to replace one's `Instance` class in the application code with such class and call `instance.migrate_2_to_3` on init.

For instance, given following umongo 3 application code

```
from umongo.frameworks.pymongo import PyMongoInstance

instance = PyMongoInstance()

# Register embedded documents
[...]

@instance.register
class Doc(Document):
    name = fields.StrField()
    # Embed documents
    embedded = fields.EmbeddedField([...])

instance.set_db(pymongo.MongoClient())

# This may raise an exception if Doc contains embedded documents
# as described above
Doc.find()
```

the migration can be performed by calling migrate_2_to_3.

```
from umongo.frameworks.pymongo import PyMongoMigrationInstance

instance = PyMongoMigrationInstance()

# Register embedded documents
```

(continues on next page)

(continued from previous page)

```
[...]

@instance.register
class Doc(Document):
    name = fields.StrField()
    # Embed documents
    embedded = fields.EmbeddedField([...])

instance.set_db(pymongo.MongoClient())
instance.migrate_2_to_3()

# This is safe now that the database is migrated
Doc.find()
```

Of course, this needs to be done only once. Although the migration is idempotent, it wouldn't make sense to keep this in the codebase and execute the migration on every application startup.

However, it is possible to embed the migration feature in the application code by defining a dedicated command, like a Flask CLI command for instance.

1.3 API Reference

1.3.1 Instance

class umongo.instance.**Instance**(*db=None*)

Abstract instance class

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

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

instance = MyFrameworkInstance()
# 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*.

register(*template*)

Generate an `umongo.template.Implementation` from the given `umongo.template.Template` for this instance.

Parameters **template** – `umongo.template.Template` to implement

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*).

set_db(*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.frameworks.pymongo.**PyMongoInstance**(*db=None*)
umongo.instance.Instance implementation for pymongo

1.3.2 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*, *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):
```

(continues on next page)

(continued from previous page)

```
    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
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*, *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*)

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.

1.3.3 EmbeddedDocument

umongo.**EmbeddedDocument**
Shortcut to EmbeddedDocumentTemplate

alias of *umongo.embedded_document.EmbeddedDocumentTemplate*

class umongo.embedded_document.**EmbeddedDocumentTemplate**(**args*, ***kwargs*)
Base class to define a umongo embedded document.

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

---

class umongo.embedded_document.**EmbeddedDocumentOpts**(*instance*, *template*, *abstract=False*, *is_child=False*, *strict=True*, *offspring=None*)
Configuration for an `umongo.embedded_document.EmbeddedDocument`.

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

```
@instance.register
class MyEmbeddedDoc(EmbeddedDocument):
    class Meta:
        abstract = True

assert MyEmbeddedDoc.opts.abstract == True
```

attribute	configurable in Meta	description
template	no	Origin template of the embedded document
instance	no	Implementation's instance
abstract	yes	Embedded document can only be inherited
is_child	no	Embedded document inherit of a non-abstract embedded document
strict	yes	Don't accept unknown fields from mongo (default: True)
offspring	no	List of embedded documents inheriting this one

class umongo.embedded_document.**EmbeddedDocumentImplementation**(***kwargs*)
Represent an embedded document once it has been implemented inside a `umongo.instance.BaseInstance`.

classmethod build_from_mongo(*data*, *use_cls=True*)
Create an embedded 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 EmbeddedDocument class to instanciate

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

dump()
Dump the embedded document.

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

1.3.4 MixinDocument

umongo.**MixinDocument**
Shortcut to MixinDocumentTemplate

alias of *umongo.mixin.MixinDocumentTemplate*

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

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

---

class umongo.mixin.**MixinDocumentOpts**(*instance*, *template*)
Configuration for an `umongo.mixin.MixinDocument`.

attribute	configurable in Meta	description
template	no	Origin template of the embedded document
instance	no	Implementation's instance

class umongo.mixin.**MixinDocumentImplementation**
Represent a mixin document once it has been implemented inside a `umongo.instance.BaseInstance`.

1.3.5 Abstracts

class umongo.abstract.**BaseSchema**(**args*, ***kwargs*)
All schema used in umongo should inherit from this base schema

MA_BASE_SCHEMA_CLS
alias of `BaseMarshmallowSchema`

as_marshmallow_schema()
Return a pure-marshmallow version of this schema 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.

MARSHMALLOW_ARGS_PREFIX = 'marshmallow_'

as_marshmallow_field()
Return a pure-marshmallow version of this field

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

deserialize_from_mongo(*value*)

serialize_to_mongo(*obj*)

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*)

1.3.6 Fields

umongo fields

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*)

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

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

as_marshmallow_field()

Return a pure-marshmallow version of this field

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

as_marshmallow_field()

Return a pure-marshmallow version of this field

map_to_field(*mongo_path*, *path*, *func*)

Apply a function to every field in the schema

class umongo.fields.**ConstantField**(**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*)

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*)

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

as_marshmallow_field()
Return a pure-marshmallow version of this field

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

1.3.7 Data objects

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

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

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

clear_modified()

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

inner_field

insert(*i*, *obj*)
Insert object before index.

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**(*key_field*, *value_field*, **args*, ***kwargs*)

clear_modified()

is_modified()

key_field

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

set_modified()

setdefault(*key*, *obj=None*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

value_field

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.

1.3.8 Marshmallow integration

Pure marshmallow fields used in umongo

class umongo.marshmallow_bonus.**ObjectId**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]], None] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Marshmallow field for bson.objectid.ObjectId

class umongo.marshmallow_bonus.**Reference**(*, *load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]], None] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)

Marshmallow field for *umongo.fields.ReferenceField*

class umongo.marshmallow_bonus.**GenericReference**(*, *load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]], None] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)

Marshmallow field for *umongo.fields.GenericReferenceField*

1.3.9 Exceptions

umongo exceptions

exception umongo.exceptions.**AbstractDocumentError**
Raised when instantiating an abstract document

exception umongo.exceptions.**AlreadyCreatedError**
Modifying id of an already created document

exception umongo.exceptions.**AlreadyRegisteredDocumentError**
Document already registerd

exception umongo.exceptions.**DeleteError**
Error while deleting document

exception umongo.exceptions.**DocumentDefinitionError**
Error in document definition

exception umongo.exceptions.**NoCompatibleInstanceError**
Can't find instance compatible with database

exception umongo.exceptions.**NoDBDefinedError**
No database defined

exception umongo.exceptions.**NoneReferenceError**
Retrieving a None reference

exception umongo.exceptions.**NotCreatedError**
Document does not exist in database

exception umongo.exceptions.**NotRegisteredDocumentError**
Document not registered

exception umongo.exceptions.**UMongoError**
Base umongo error

exception umongo.exceptions.**UnknownFieldInDBError**
Data from database contains unknown field

exception umongo.exceptions.**UpdateError**
Error while updating document

1.4 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:

1.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/Scille/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 :)

1.4.2 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.

1.4.3 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
```

1.4.4 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 3.7 and 3.8. Check https://travis-ci.org/touilleMan/umongo/pull_requests and make sure that the tests pass for all supported Python versions.

1.5 Credits

1.5.1 Original Author

- Emmanuel Leblond @touilleMan

1.5.2 Development Lead

- Jérôme Lafréchoux @lafrech

1.5.3 Contributors

- Walter Scheper @wfscheper
- Imbolc @imbolc
- @patlach42
- Serj Shevchenko @serjshevchenko
- Élysson MR @elyssonmr
- Mandar Upadhye @mandarup
- Pavel Kulyov @pkulev

- Felix Sonntag @fsonntag
- Attila Kóbor @atti92
- Denis Moskalets @denya
- Phil Chiu @whophil

1.6 History

1.6.1 3.1.0 (2021-12-23)

Features:

- Add fields list to `Document` and `EmbeddedDocument.__dir__()` (see #367).

Bug fixes:

- Test database by comparing to `None`, not casting to `bool` to prevent an exception raised by pymongo >= 4 (see #366).

1.6.2 3.0.1 (2021-10-16)

Bug fixes:

- Fix ListField.insert: trigger `set_modified`, deserialize using inner field (see #364).

1.6.3 3.0.0 (2021-01-11)

Features:

- Fix internationalization of generated marshmallow fields for container fields (DictField, ListField, NestedField) (see #329).
- Don't pass field metadata as kwargs (deprecated in marshmallow 3.10.0) but as `metadata` argument (see #328).

Bug fixes:

- Fix IO validation of *None* for references, lists and embedded documents (see #330).
- Add *_dict_io_validate* to propagate IO validation through *DictField* (see #335).

Other changes:

- *Backwards-incompatible*: Require marshmallow>=3.10.0 (see #328).

1.6.4 3.0.0b14 (2020-12-08)

Features:

- Provide `Instance` subclasses for each framework to help users migrating a database from umongo 2 to umongo 3 (see #319).

- *Backwards-incompatible*: Postpone embedded document resolution in `EmbeddedField` to allow passing an embedded document as string before its registration. Unknown embedded document errors in `EmbeddedField` are now detected at runtime, not registration time. Also, indexes are now collected on first use rather than upon registration and should be accesses through `Document.indexes` cached property rather than `Document.opts.indexes`. (see #322)
- *Backwards-incompatible*: Make `BaseSchema` ordered. This fixes querying on embedded documents. Make `BaseMarshmallowSchema` ordered as well. (see #323)
- *Backwards-incompatible*: Make `RemoveMissingSchema` opt-out. By default, generated pure marshmallow schemas now skip missing values from `Document` instances rather that returning `None`. This can be changed by setting `MA_BASE_SCHEMA_CLS`. (see #325)

1.6.5 3.0.0b13 (2020-11-23)

Bug fixes:

- Fix a bug introduced in 3.0.0b12 preventing instance initialization with DB as parameter as in `instance = PyMongoInstance(db)`. (see #318)

1.6.6 3.0.0b12 (2020-11-16)

Features:

- *Backwards-incompatible*: Rework `Instance`: merge `BaseInstance`, `Instance` and `LazyLoaderInstance` into a single abstract `Instance` class. Remove `templates` argument from `Instance`. Rename `Instance.init` to `Instance.set_db`. Don't republish concrete framework instances in `umongo` top module. (see #314)
- Add `session` context manager to `PyMongoInstance` and `MotorAsyncIOInstance`. This allows to use session related features (causally consistent reads, transactions) from umongo. (see #315)

1.6.7 3.0.0b11 (2020-11-06)

Features:

- *Backwards-incompatible*: Allow setting arbitrary attributes on `Document` and `EmbeddedDocument` instances. This change is part of a refactor meant to simplify set / get / delete operations on document objets and (marginally) improve performance. (see #272)
- Use structured information provided with `DuplicateKeyError` rather than parse the error message string (see #309).
- Add `replace` argument to `commit` method to force writing the whole document rather than updating (see #310).

Other changes:

- Support Python 3.9 (see #311).
- *Backwards-incompatible*: Drop motor<2.0.0 support (see #312).
- *Backwards-incompatible*: Drop MongoDB<4.2 support (see #313).

1.6.8 3.0.0b10 (2020-10-12)

Features:

- Allow passing `Document` and `EmbeddedDocument` in queries. (see #303)

1.6.9 3.0.0b9 (2020-10-05)

Features:

- Add support for motor 2.2 (see #294). (Picked from 2.3.0.)
- *Backwards-incompatible*: Add `ExposeMissing` context manager to return `missing` rather than `None` when dumping. Replace `FromUmongoSchema` with `RemoveMissingSchema`. This schema removes missing fields when dumping by using `ExposeMissing` internally. Make this feature opt-in by requiring the user to specify `RemoveMissingSchema` as `MA_BASE_SCHEMA_CLS`. (see #261)
- *Backwards-incompatible*: Remove `mongo_world` parameter from `Schema.as_marshmallow_schema`. Schemas generated by this method are meant to (de)serialize umongo objects, not `dict` straight from database. (see #299)
- *Backwards-incompatible*: Remove `umongo.Schema`. Schemas should inherit from `umongo.abstract.BaseSchema`. Expose `RemoveMissingSchema` as `umongo.RemoveMissingSchema`. (see #301)

Other changes:

- *Backwards-incompatible*: Drop Python 3.6 support (see #298).

1.6.10 3.0.0b8 (2020-07-22)

Features:

- Let `Document` inherit from `EmbeddedDocument` (see #266).
- Add `MixinDocument` allowing `Document` and `EmbeddedDocument` to inherit fields and pre/post methods from mixin objects (see #278).
- *Backwards-incompatible*: Remove `as_attribute` argument of `BaseInstance.register` method. Documents can not be accessed by name as instance attributes anymore. (see #290)

Bug fixes:

- Fix passing `None` to a field with `_required_validate` method (see #289).

1.6.11 3.0.0b7 (2020-05-08)

Features:

- *Backwards-incompatible*: Revert broken feature introduced in 3.0.0b6 allowing to get fields from mixin classes (see #273).
- *Backwards-incompatible*: Remove `allow_inheritance` option. Any `Document` or `EmbeddedDocument` may be subclassed (see #270).
- *Backwards-incompatible*: `Field` raises `DocumentDefinitionError` rather than `RuntimeError` when passed `missing` kwarg and `Document.commit` raises `NotCreatedError` when passed `conditions` for a document that is not in database (see #275).

1.6.12 3.0.0b6 (2020-05-04)

Features:

- *Backwards-incompatible*: `abstract` in `EmbeddedDocument` behaves consistently with `Document`. The `_cls` / `cls` field is only added on concrete embedded documents subclassing concrete embedded documents. And `EmbeddedField` only accepts concrete embedded documents. (see #86)
- `Document` and `EmbeddedDocument` may inherits from mixin classes. The mixin class should appear first (leftmost) in the bases: `class MyDocument(MyMixin, Document)`. (see #188)

Other changes:

- *Backwards-incompatible*: marshmallow imports throughout the code are done as `import marshmallow as ma`. For convenience, `missing` and `ValidationError` can still be imported as `umongo.missing` and `umongo.ValidationError`.

1.6.13 3.0.0b5 (2020-04-30)

Features:

- *Backwards-incompatible*: Add `MA_BASE_SCHEMA_CLS` class attribute to `Document` and `EmbeddedDocument` to specify a base class to use in `as_marshmallow_schema`. Drop the `check_unknown_fields`, `params` and `meta` attributes of `as_marshmallow_schema`. Make `mongo_world` kwarg-only. The same effect can be achieved using base schema classes. This incidentally fixes broken `as_marshmallow_schema` cache feature. (see #263)
- *Backwards-incompatible*: Add `TxMongoDocument.find_with_cursor` and drop support for upstream deprecated `find(cursor=True)`. (see #259).

Other changes:

- *Backwards-incompatible*: Require txmongo>=19.2.0 (see #259).

1.6.14 3.0.0b4 (2020-04-27)

Features:

- *Backwards-incompatible*: Remove partial load feature (see #256).
- *Backwards-incompatible*: Add `Document.pk_field` and remove `BaseDataProxy.*_by_mongo_name methods` (see #257).
- *Backwards-incompatible*: Raise AlreadyCreatedError when modifying pk of created document (see #258).

1.6.15 3.0.0b3 (2020-04-26)

Features:

- *Backwards-incompatible*: Replace `ReferenceError` with `NoneReferenceError`. Review the list of exceptions importable from root `umongo` module. (see #251)

Bug fixes:

- Don't modify data when calling `set_by_mongo_name` on a field that was not loaded in a partial load. (see #253)

Other changes:

- *Backwards-incompatible*: Drop Python 3.5 support (see #248).

1.6.16 3.0.0b2 (2020-04-18)

Features:

- Use fields for keys/values in DictField (see #245).

Bug fixes:

- Fix BaseField.__repr__ (see #247).

1.6.17 3.0.0b1 (2020-03-29)

Features:

- Support marshmallow 3 (see #154).
- All field parameters beginning with `"marshmallow_"` are passed to the marshmallow schema, rather than only a given list of known parameters. (see #228)

Other changes:

- *Backwards-incompatible*: Drop support for marshmallow 2. See marshmallow upgrading guide for a comprehensive list of changes. (see #154)
- *Backwards-incompatible*: `StrictDateTimeField` is removed as marshmallow now provides `NaiveDateTimeField` and `AwareDateTimeField`. (see #154)
- *Backwards-incompatible*: `default` shall now be provided in deserialized form. (see #154)

1.6.18 2.3.0 (2020-09-06)

Features:

- Add support for motor 2.2 (see #294).

1.6.19 2.2.0 (2019-12-18)

Bug fixes:

- Fix `find`/`find_one`: pass `filter` as first positional argument (see #215).

Other changes:

- Support Python 3.8 (see #210).

1.6.20 2.1.1 (2019-10-04)

Bug fixes:

- Fix `ObjectId` bonus field: catch `TypeError` when deserializing (see #207).

1.6.21 2.1.0 (2019-06-19)

Features:

- Add support for motor 2.+ by adding a `count_documents` class method to the `MotorAsyncIODocument` class. `count_documents` attempts to transparently use the correct motor call signature depending on which version of the driver is installed. Note that the behavior of the cursor object returned by `MotorAsyncIODocument.find` strictly adheres to the interface provided by the underlying driver.

1.6.22 2.0.5 (2019-06-13)

Bug fixes:

- Ensure `Reference` and `GenericReference` fields round-trip (see #200).

1.6.23 2.0.4 (2019-05-28)

Bug fixes:

- Include modified `BaseDataObject` in `BaseDataProxy.get_modified_fields` and `BaseDataProxy.get_modified_fields_by_mongo_name` (see #195).
- Always return a boolean in `List.is_modified` (see #195).
- `List`: call `set_modified` when deleting an element using the `del` builtin (see #195).

1.6.24 2.0.3 (2019-04-10)

Bug fixes:

- Fix millisecond overflow when milliseconds round to 1s in `StrictDateTimeField` (see #189).

1.6.25 2.0.2 (2019-04-10)

Bug fixes:

- Fix millisecond overflow when milliseconds round to 1s in `DateTimeField` and `LocalDateTimeField` (see #189).

1.6.26 2.0.1 (2019-03-25)

Bug fixes:

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

1.6.27 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).

1.6.28 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)

1.6.29 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).

1.6.30 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)

1.6.31 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)

1.6.32 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)

1.6.33 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)

1.6.34 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

1.6.35 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

1.6.36 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)

1.6.37 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

1.6.38 0.8.1 (2016-05-19)

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

1.6.39 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

1.6.40 0.7.8 (2016-4-28)

- Fix setup.py style preventing release of version 0.7.7

1.6.41 0.7.7 (2016-4-28)

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

1.6.42 0.7.6 (2016-4-28)

- Use extras_require to install driver along with umongo

1.6.43 0.7.5 (2016-4-23)

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

1.6.44 0.7.4 (2016-4-21)

- Fix missing package in setup.py

1.6.45 0.7.3 (2016-4-21)

- Fix setup.py style preventing from release

1.6.46 0.7.2 (2016-4-21)

- Fix crash when generating indexes on EmbeddedDocument

1.6.47 0.7.1 (2016-4-21)

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

1.6.48 0.7.0 (2016-4-21)

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

1.6.49 0.6.1 (2016-4-13)

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

1.6.50 0.6.0 (2016-4-12)

- Heavy improvements everywhere !

1.6.51 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