nanomongo Documentation

Release 0.1

Eren Güven

Apr 02, 2018

Contents

nanomongo is a minimal MongoDB Object-Document Mapper for Python. It does not attempt to be a feature-complete ODM but if you like using `pymongo` api with python dictionaries and often find yourself writing validators and `pymongo.Collection` wrappers, nanomongo might suit your needs.

nanomongo has full test coverage.

Quick Links: Source (github) - Documentation (rtd) - Packages (PyPi)

Version 0.4: Utility methods *dbref_field_getters*, `get_dbref()` and Bugfix Python23 text type compatibility

Version 0.3: nanomongo is now python2 compatible (with syntactic difference when defining your Document, see *Defining Your Document* below).

CHAPTER 1

Installation

```
$ pip install nanomongo
```

CHAPTER 2

Quick Tutorial

2.1 Defining Your Document

```
import pymongo
from nanomongo import Index, Field, BaseDocument

mclient = pymongo.MongoClient()

class Py23CompatibleDoc(BaseDocument):
    client = mclient
    db = 'dbname'
    dot_notation = True
    foo = Field(str)
    bar = Field(int, required=False)

# Python3 only
class Py3Doc(BaseDocument, dot_notation=True, client=mclient, db='dbname'):
    foo = Field(str)
    bar = Field(int, required=False)

    __indexes__ = [
        Index('foo'),
        Index([('bar', 1), ('foo', -1)], unique=True),
    ]
```

You don't have to declare `client` or `db` as shown above, you can *`register()`* (and I definitely prefer it on python2) your document later as such:

```
client = pymongo.MongoClient()
MyDoc.register(client=client, db='dbname', collection='mydoc')
```

If you omit `collection` when defining/registering your document, `__name__.lower()` will be used by default

2.2 Creating, Inserting, Saving

```
doc = MyDoc(foo='42')  # or MyDoc({'foo': '42'})
doc.bar = 42  # attribute style access because dot_notation=True
doc.insert()
```

insert() is a wrapper around `pymongo.Collection().insert()` and has the same return value (`_id`) unless you explicitly set `w=0`

```
doc.foo = 'new foo'  # this change is recorded
del doc.bar  # this is recored as well
doc.save()  # save only does partial updates
```

save() uses `pymongo.Collection().update()` with the changed data. The above will run

```
update({'_id': doc['_id']}, {'$set': {'foo': 'new foo'}, '$unset': {'bar': 1}})
```

2.3 Querying

```
Doc.find({'bar': 42})
Doc.find_one({'foo': 'new foo'})
```

find() and *find_one()* methods are essentially wrappers around respective methods of `pymongo.Collection()` and they take the same arguments.

CHAPTER 3

Extensive Example

See example

CHAPTER 4

Advanced Features

4.1 $addToSet

MongoDB `$addToSet` update modifier is very useful. nanomongo implements it.

`addToSet()` will do the *add-to-field-if-doesnt-exist* on your document instance and record the change to be applied later when *`save()`* is called.

```
# lets expand our MyDoc
class NewDoc(MyDoc):
    list_field = Field(list)
    dict_field = Field(dict)

NewDoc.register(client=client, db='dbname')
doc_id = NewDoc(list_field=[42], dict_field={'foo':[]}).insert()
doc = NewDoc.find_one({'_id': doc_id})

doc.addToSet('list_field', 1337)
doc.addToSet('dict_field.foo', 'like a boss')
doc.save()
```

Both of the above `addToSet` are applied to the `NewDoc` instance like MongoDB does it eg.

- create list field with new value if it doesn't exist
- add new value to list field if it's missing (append)
- complain if it is not a list field

When save is called, query becomes:

```
update({'$addToSet': {'list_field': {'$each': [1337]}},
                      'dict_field.foo': {'$each': ['like a boss']}})
```

Undefined fields or field type mismatch raises *`ValidationError`*.

4.2 QuerySpec check

find() and *find_one()* has a simple check against queries that can not match, logging warnings. This is an experimental feature at the moment and only does type checks as such:

`{'foo': 1}` will log warnings if

- Document has no field named `foo` (field existence)
- `foo` field is not of type `int` (field data type)

or `{'foo.bar': 1}` will log warnings if

- `foo` field is not of type `dict` or `list` (dotted field type)

CHAPTER 5

dbref_field_getters

Documents that define `bson.DBRef` fields automatically generate getter methods through *nanomongo.document.ref_getter_maker()* where the generated methods have names such as `get_<field_name>_field`.

```
class MyDoc(BaseDocument):
    # document_class with full path
    source = Field(DBRef, document_class='some_module.Source')
    # must be defined in same module as this will use
    # mydoc_instance.__class__.__module__
    destination = Field(DBRef, document_class='Destination')
    # autodiscover
    user = Field(DBRef)
```

nanomongo tries to guess the `document_class` if it's not provided by looking at registered subclasses of *BaseDocument*. If it matches two (for example when two document classes use the same collection), it will raise *UnsupportedOperation*.

CHAPTER 6

pymongo & motor

Throughout the documentation, `pymongo` is referenced but all features work the same when using motor transparently if you register the document class with a `motor.MotorClient`.

```
import motor
from nanomongo import Field, BaseDocument

class MyDoc(BaseDocument, dot_notation=True):
    foo = Field(str)
    bar = Field(list, required=False)

client = motor.MotorClient().open_sync()
MyDoc.register(client=client, db='dbname')

# and now some async motor queries (using @gen.engine)
doc_id = yield motor.Op(MyDoc(foo=42).insert)
doc = yield motor.Op(MyDoc.find_one, {'foo': 42})
doc.addToSet('bar', 1337)
yield motor.Op(doc.save)
```

Note however that pymongo vs motor behaviour is not necessarily identical. Asynchronous methods require `tornado.ioloop.IOLoop`. For example, *`register()`* runs `ensure_index` but the query will not be sent to MongoDB until `IOLoop.start()` is called.

CHAPTER 7

Contents

7.1 nanomongo.field — Field

7.1.1 Field

class nanomongo.field.**Field**(**args*, ***kwargs*)

Instances of this class is used to define field types and automatically create validators. Note that a Field definition has no value added:

```
field_name = Field(str, default='cheeseburger')
foo = Field(datetime, auto_update=True)
bar = Field(list, required=False)
```

classmethod check_kwargs(*kwargs*, *data_type*)

Check keyword arguments & their values given to `Field` constructor such as `default`, `required` ...

generate_validator(*t*, ***kwargs*)

Generates and returns validator function `(value_to_check, field_name='')`. `field_name` kwarg is optional, used for better error reporting.

`Field.`**`__init__`**(**args*, ***kwargs*)

Field kwargs are checked for correctness and field validator is set, along with other attributes such as `required` and `auto_update`

Keyword Arguments

- *default*: default field value, must pass type check, can be a `callable`
- *required*: if `True` field must exist and not be `None` (default: `True`)
- *auto_update*: set value to `datetime.utcnow()` before inserts/saves; only valid for datetime fields (default: `False`)

7.2 nanomongo.document — BaseDocument

7.2.1 BaseDocument

class nanomongo.document.**BaseDocument**(**args*, ***kwargs*)
BaseDocument class. Subclasses should be used. See *__init__()*

BaseDocument.**__init__**(**args*, ***kwargs*)
Inits the document with given data and validates the fields (field validation bad idea during init?). If you define __init__ method for your document class, make sure to call this

```
class MyDoc(BaseDocument, dot_notation=True):
    foo = Field(str)
    bar = Field(int, required=False)

    def __init__(self, *args, **kwargs):
        super(MyDoc, self).__init__(*args, **kwargs)
        # do other stuff
```

classmethod BaseDocument.**register**(*client=None*, *db=None*, *collection=None*)
Register this document. Sets client, database, collection information, builds (ensure) indexes and sets SON manipulator

classmethod BaseDocument.**get_collection**()
Returns collection as set in nanomongo

classmethod BaseDocument.**find**(**args*, ***kwargs*)
pymongo.Collection().find wrapper for this document

classmethod BaseDocument.**find_one**(**args*, ***kwargs*)
pymongo.Collection().find_one wrapper for this document

BaseDocument.**validate**()
Override this to add extra document validation, will be called at the end of *validate_all()*

BaseDocument.**validate_all**()
Check against extra fields, run field validators and user-defined *validate()*

BaseDocument.**validate_diff**()
Check correctness of diffs before partial update, also run user-defined *validate()*

BaseDocument.**run_auto_updates**()
Runs functions in nanomongo.transforms like auto_update stuff before *insert() save()*

BaseDocument.**insert**(***kwargs*)
Insert document into database, return _id. Runs *run_auto_updates()* and *validate_all()*

BaseDocument.**save**(***kwargs*)
Saves document. This method only does partial updates and no inserts. Runs *run_auto_updates()* and *validate_all()* prior to save. Returns Collection.update() response

BaseDocument.**addToSet**(*field*, *value*)
MongoDB Collection.update() $addToSet functionality. This sets the value accordingly and records the change in __nanodiff__ to be saved with *save()*.

```
# MongoDB style dot notation can be used to add to lists
# in embedded documents
doc = Doc(foo=[], bar={})
doc.addToSet('foo', new_value)
doc.addToSet('bar.sub_field', new_value)
```

Contrary to how $set has no effect under __setitem__ (see *RecordingDict*.__setitem__) when the new value is equal to the current; $addToSet explicitly adds the call to __nanodiff__ so it will be sent to the database when *save()* is called.

BaseDocument.**get_dbref**()
create a bson.DBRef instance for this *BaseDocument* instance

nanomongo.document.**ref_getter_maker**(*field_name*, *document_class=None*)
create dereference methods for given field_name to be bound to Document instances

7.3 nanomongo.util — RecordingDict

7.3.1 RecordingDict

class nanomongo.util.**RecordingDict**(**args*, ***kwargs*)
A dictionary subclass modifying __setitem__() and __delitem__() methods to record changes in its __nanodiff__ attribute

check_can_update(*modifier*, *field_name*)
Check if given *modifier field_name* combination can be added. MongoDB does not allow field duplication with update modifiers. This is to be used with methods *addToSet* ...

clean_other_modifiers(*current_mod*, *field_name*)
Given *current_mod*, removes other *field_name* modifiers, eg. when called with *$set*, removes *$unset* and *$addToSet* etc. on *field_name*

get_sub_diff()
get *__nanodiff__* from embedded documents. Find fields of *RecordingDict* type, iterate over their diff and build dotted keys for top level diff

reset_diff()
reset *__nanodiff__* recursively; to be used after saving diffs. This does NOT do a rollback. Reload from db for that

RecordingDict.**__init__**(**args*, ***kwargs*)

class nanomongo.util.**NanomongoSONManipulator**(*as_class*, *transforms=None*)
A pymongo SON Manipulator used on data that comes from the database to transform data to the document class we want because *as_class* argument to pymongo find methods is called in a way that screws us.

- Recursively applied, we don't want that
- *__init__* is not properly used but rather __setitem__, fails us

JIRA: PYTHON-175 PYTHON-215

7.4 nanomongo.errors

class nanomongo.errors.**NanomongoError**
Base nanomongo error

class nanomongo.errors.**ValidationError**
Raised when a field fails validation

class nanomongo.errors.**ExtraFieldError**
Raised when a document has an undefined field

class nanomongo.errors.**ConfigurationError**
Raised when a required value found to be not set during operation, or a Document class is registered more than once

class nanomongo.errors.**IndexMismatchError**
Raised when a defined index does not match defined fields

class nanomongo.errors.**UnsupportedOperation**
Raised when an unsupported opperation/parameters are used

class nanomongo.errors.**DBRefNotSetError**
Raised when a DBRef getter is called on not-set DBRef field

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

B

C

D

E

F

G

I

N

R

S

U

V