
nanomongo Documentation

Release 0.1

Eren Güven

Apr 03, 2018

Contents

1	Installation	3
2	Quickstart	5
2.1	Defining A Document And Registering	5
2.2	Creating, Inserting, Querying, Saving	6
3	Extensive Example	7
3.1	Extensive Example	7
4	Advanced Features	11
4.1	\$addToSet	11
4.2	QuerySpec check	12
4.3	dbref_field_getters	12
5	pymongo & motor	13
6	Contents	15
6.1	nanomongo.document	15
6.2	nanomongo.errors	16
6.3	nanomongo.field	17
6.4	nanomongo.util	17
7	Indices and tables	19
	Python Module Index	21

nanomongo is a minimal MongoDB Object-Document Mapper for Python. It does not attempt to be a feature-complete ODM but if you enjoy using [PyMongo](#) API with dictionaries and often find yourself writing validators and `pymongo.Collection` wrappers, nanomongo might suit your needs.

Quick Links: [Source \(github\)](#) - [Documentation \(rtd\)](#) - [Packages \(PyPi\)](#) - [Changelog](#)

CHAPTER 1

Installation

```
$ pip install nanomongo
```


2.1 Defining A Document And Registering

You can define a document as shown below:

```
import pymongo
from nanomongo import Field, BaseDocument

class Py23Doc(BaseDocument):
    dot_notation = True # to allow attribute-like access to document keys
    foo = Field(str)
    bar = Field(int, required=False)

    __indexes__ = [
        pymongo.IndexModel('foo'),
        pymongo.IndexModel([('bar', 1), ('foo', -1)], unique=True),
    ]

# before use, the document needs to be registered. The following will connect
# to the database and create indexes if necessary
Py23Doc.register(client=pymongo.MongoClient(), db='mydbname', collection='Py23Doc')
```

Python3 allows slightly cleaner definitions:

```
# Python3 only
class MyDoc(BaseDocument, dot_notation=True):
    foo = Field(str)
    bar = Field(int, required=False)
```

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

2.2 Creating, Inserting, Querying, Saving

```
doc = MyDoc(foo='1337', bar=42) # creates document {'foo': '1337', 'bar': 42}
doc.insert() # returns pymongo.results.InsertOneResult
MyDoc.find_one({'foo': '1337'}) # returns document {'_id': ObjectId('...'), 'bar': 42, 'foo': '1337'}

doc.foo = '42' # records the change
del doc.bar # records the change
# save only does partial updates, this will call
# collection.update_one({'_id': doc['_id']}, {'$set': {'foo': '42'}, '$unset': {'bar': 1}})
doc.save() # returns pymongo.results.UpdateResult

MyDoc.find_one({'foo': '1337'}) # returns None
MyDoc.find_one({'foo': '42'}) # returns document {'_id': ObjectId('...'), 'foo': '42'}
```

`insert()` is a wrapper around `pymongo.Collection.insert_one()` and `save()` is a wrapper around `pymongo.Collection.update_one()`. They pass along received keyword arguments and have the same return value.

`find()` and `find_one()` methods are wrappers around respective methods of `pymongo.Collection` with same arguments and return values.

3.1 Extensive Example

Following is an example use case.

```
from datetime import datetime

import bson
import pymongo
import six

from nanomongo import Field, BaseDocument

class User(BaseDocument):
    """A user has a name, a list of categories he follows and a dictionary
    for preferences.

    We index on :attr:`~User.name` field and on
    :attr:`~User.following` + :attr:`~preferences.notifications` (compound),
    think of listing followers of a category who have notifications enabled.
    """
    dot_notation = True
    name = Field(six.text_type)
    following = Field(list, default=[])
    preferences = Field(dict, default={'notifications': True})

    __indexes__ = [
        pymongo.IndexModel('name'),
        pymongo.IndexModel([
            ('following', pymongo.ASCENDING),
            ('preferences.notifications', pymongo.ASCENDING)
        ]),
    ]
```

```

def add_entry(self, title, categories=None):
    """Add an entry with title and categories and `user=self._id`"""
    assert (title and isinstance(title, six.text_type)), 'title not str or empty'
    e = Entry(user=self._id, title=title)
    if categories:
        assert isinstance(categories, list), 'categories not a list'
        for cat in categories:
            assert (cat and isinstance(cat, six.text_type)), 'categories element_
↳not str or empty'
            e.categories = categories
    e.insert()
    return e

def follow(self, *categories):
    """Start following a category (add it to :attr:`~self.categories`)"""
    assert categories, 'categories expected'
    for category in categories:
        assert (category and isinstance(category, six.text_type)), 'category not_
↳str or empty'
        self.add_to_set('following', category)
    self.save()

def get_entries(self, **kwargs):
    """Get entries (well cursor for them) of this User, extra kwargs
    (such as limit) are passed to :class:`~pymongo.Collection().find()`
    """
    cursor = Entry.find({'user': self._id}, **kwargs)
    # hint not necessary here, just demonstration
    cursor.hint([('user', pymongo.ASCENDING), ('_id', pymongo.DESCENDING)])
    return cursor

def get_comments(self, with_entries=False, **kwargs):
    """Get comments of this User, extra kwargs
    (such as limit) are passed to :class:`~pymongo.Collection().find()`
    of :class:`~Entry`. Default gets just the comments, ``with_entries=True``
    to get entries as well. Returns generator
    """
    cursor = Entry.find({'comments.author': self.name}, **kwargs)
    if with_entries:
        for entry in cursor:
            yield entry
    for entry in cursor:
        for comment in entry.comments:
            if self.name == comment['author']:
                yield comment

class Entry(BaseDocument):
    """An entry that a :class:`~User` posts; has a title, a user field
    pointing to a User _id, a list of categories that the entry belongs
    and a list for comments.

    We index on categories, 'comments.author' + 'comment.created'
    so we can lookup comments by author and
    'user' + '_id' so we can chronologically sort entries by user
    """
    dot_notation = True
    user = Field(bson.ObjectId)

```

```
title = Field(six.text_type)
categories = Field(list, default=[])
comments = Field(list, default=[])

__indexes__ = [
    pymongo.IndexModel([('user', pymongo.ASCENDING), ('_id', pymongo.
↪DESCENDING)]),
    pymongo.IndexModel('categories'),
    pymongo.IndexModel([('comments.author', pymongo.ASCENDING), ('comments.created
↪', pymongo.DESENDING)]),
]

def add_comment(self, text, author):
    """Add a comment to this Entry"""
    assert (text and isinstance(text, six.text_type)), 'text not str or empty'
    assert (author and isinstance(author, User)), 'second argument not an_
↪instance of User'
    doc = {'text': text, 'author': author.name, 'created': datetime.utcnow()}
    # TODO: push is more appropriate in this situation, add when implemented
    self.add_to_set('comments', doc)
    # we could have also done self.comments = self.comments + [doc]
    self.save()
    return text

def get_followers(self):
    """Return a cursor for Users who follow the categories that this Entry has
    """
    return User.find({'following': {'$in': self.categories}})
```


4.1 \$addToSet

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

`add_to_set()` 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.

```
import pymongo
from nanomongo import Field, BaseDocument

class NewDoc(BaseDocument, dot_notation=True):
    list_field = Field(list)
    dict_field = Field(dict)

NewDoc.register(client=pymongo.MongoClient(), db='mydbname')
doc_id = NewDoc(list_field=[42], dict_field={'foo':[]}).insert().inserted_id
doc = NewDoc.find_one({'_id': doc_id})
# {'_id': ObjectId('...'), 'dict_field': {'foo': []}, 'list_field': [42]}

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

Both of the above `add_to_set` calls 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, the following is called:

```
update_one(
    {'_id': doc['_id']},
```

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

Undefined fields or field type mismatch raises *ValidationError*:

```
doc.add_to_set('dict_field.foo', 'like a boss')
ValidationError: Cannot apply $addToSet modifier to non-array: dict_field=<class 'dict'
↪ '>
```

4.2 QuerySpec check

find() and *find_one()* runs a simple check against queries and logs warnings for queries that can not match. See *check_spec()* for details.

4.3 dbref_field_getters

Documents that define `bson.DBRef` fields automatically generate getter methods through `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 more than one (for example when two document classes use the same collection), it will raise *UnsupportedOperation*.

0.5.0 update: motor support is currently not in a working state, this section is kept for reference.

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.add_to_set('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.

6.1 nanomongo.document

class nanomongo.document.**BaseDocument** (*args, **kwargs)

BaseDocument class. Subclasses should be used. See `__init__()`

add_to_set (field, value)

Explicitly defined \$addToSet functionality. This sets/updates the field value accordingly and records the change to be saved with `save()`.

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

Contrary to how \$set ing the same value has no effect under `__setitem__` (see `.util.RecordingDict.__setitem__()`) when the new value is equal to the current, this explicitly records the change so it will be sent to the database when `save()` is called.

classmethod find (*args, **kwargs)

`pymongo.Collection().find` wrapper for this document

classmethod find_one (*args, **kwargs)

`pymongo.Collection().find_one` wrapper for this document

classmethod get_collection ()

Returns collection as set in nanomongo

get_dbref ()

Return a `bson.DBRef` instance for this *BaseDocument* instance

insert (**kwargs)

Runs auto updates, validates the document, and inserts into database. Returns `pymongo.results.InsertOneResult`.

classmethod register (*client=None, db=None, collection=None*)

Register this document. Sets client, database, collection information, creates indexes and sets SON manipulator

run_auto_updates ()

Runs auto_update functions in `.nanomongo.transforms`.

save (***kwargs*)

Runs auto updates, validates the document, and saves the changes into database. Returns `pymongo.results.UpdateResult`.

validate ()

Override this to add extra document validation. It will be called during `insert()` and `save()` before the database operation.

validate_all ()

Check correctness of the document before `insert()`. Ensure that

- no extra (undefined) fields are present
- field values are of correct data type
- required fields are present

validate_diff ()

Check correctness of diffs (ie. `$set` and `$unset`) before `save()`. Ensure that

- no extra (undefined) fields are present for either set or unset
- field values are of correct data type
- required fields are not unset

6.2 nanomongo.errors

exception nanomongo.errors.NanomongoError

Base nanomongo error

exception nanomongo.errors.ValidationError

Raised when a field fails validation

exception nanomongo.errors.ExtraFieldError

Raised when a document has an undefined field

exception nanomongo.errors.ConfigurationError

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

exception nanomongo.errors.IndexMismatchError

Raised when a defined index does not match defined fields

exception nanomongo.errors.UnsupportedOperation

Raised when an unsupported operation/parameters are used

exception nanomongo.errors.DBRefNotSetError

Raised when a DBRef getter is called on not-set DBRef field

6.3 nanomongo.field

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

Instances of this class is used to define field types and automatically create validators:

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

__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)

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.

6.4 nanomongo.util

nanomongo.util.**check_spec**(cls, spec)

Check the query spec for given class and log warnings. Not extensive, helpful to catch mistyped queries.

- Dotted keys (eg. {'foo.bar': 1}) in spec are checked for top-level (ie. foo) field existence
- Dotted keys are also checked for their top-level field type (must be dict or list)
- Normal keys (eg. {'foo': 1}) in spec are checked for top-level (ie. foo) field existence
- Normal keys with non-dict queries (ie. not something like {'foo': {'\$gte': 0, '\$lte': 1}}) are also checked for their data type

class nanomongo.util.**RecordingDict**(*args, **kwargs)

A dict subclass modifying dict.__setitem__() and dict.__delitem__() methods to record changes internally 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 `add_to_set()` ...

clear_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()

Find fields of *RecordingDict* type, iterate over their diff and build dotted keys to be merged into 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.

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

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

n

`nanomongo.document`, 15

`nanomongo.errors`, 16

`nanomongo.field`, 17

`nanomongo.util`, 17

Symbols

`__init__()` (nanomongo.field.Field method), 17

A

`add_to_set()` (nanomongo.document.BaseDocument method), 15

B

BaseDocument (class in nanomongo.document), 15

C

`check_can_update()` (nanomongo.util.RecordingDict method), 17

`check_kwargs()` (nanomongo.field.Field class method), 17

`check_spec()` (in module nanomongo.util), 17

`clear_other_modifiers()` (nanomongo.util.RecordingDict method), 17

ConfigurationError, 16

D

DBRefNotSetError, 16

E

ExtraFieldError, 16

F

Field (class in nanomongo.field), 17

`find()` (nanomongo.document.BaseDocument class method), 15

`find_one()` (nanomongo.document.BaseDocument class method), 15

G

`generate_validator()` (nanomongo.field.Field method), 17

`get_collection()` (nanomongo.document.BaseDocument class method), 15

`get_dbref()` (nanomongo.document.BaseDocument method), 15

`get_sub_diff()` (nanomongo.util.RecordingDict method), 17

I

IndexMismatchError, 16

`insert()` (nanomongo.document.BaseDocument method), 15

N

nanomongo.document (module), 15

nanomongo.errors (module), 16

nanomongo.field (module), 17

nanomongo.util (module), 17

NanomongoError, 16

NanomongoSONManipulator (class in nanomongo.util), 18

R

RecordingDict (class in nanomongo.util), 17

`register()` (nanomongo.document.BaseDocument class method), 15

`reset_diff()` (nanomongo.util.RecordingDict method), 17

`run_auto_updates()` (nanomongo.document.BaseDocument method), 16

S

`save()` (nanomongo.document.BaseDocument method), 16

U

UnsupportedOperation, 16

V

`validate()` (nanomongo.document.BaseDocument method), 16

`validate_all()` (nanomongo.document.BaseDocument method), 16

`validate_diff()` (nanomongo.document.BaseDocument method), 16

ValidationError, 16