dongo Documentation

Release 0.4.0

Johan Nestaas

May 12, 2018

Contents

A Django-ORM inspired Mongo ODM.

Python 2.7 and 3.x compatible, and requires the database server running at least MongoDB 2.6, as PyMongo does.

CHAPTER 1

Installation

From PyPI:

```
$ pip install dongo
```

From the project root directory:

```
$ python setup.py install
```

CHAPTER 2

Usage

Dongo is a Django-ORM inspired ODM for mongodb.

Here are a few examples of the query and class syntax.

You will need to first connect to a database and host. By default, localhost port 27017 will be selected, but you will still need to specify the default database:

```
from dongo import connect

# For the mydatabase named database on localhost
connect('mydatabase')

# For your mongodb in the private network
connect('mydatabase', host='192.168.1.200')

# For multiple hosts in a replica set
connect('mydatabase', hosts=['10.0.0.100', '10.0.0.101', '10.0.0.102:27018'], replica_
↪set='myrepset0')

# A uri can explicitly be specified as well
connect('mydatabase', uri='mongodb://localhost:27017/')
```

You can separate collections into different databases, but those connections select the default database that collections will use if database is unspecified.

Next you will need to declare some sort of collection classes:

```
from dongo import DongoCollection
from datetime import datetime

class MusicArtist(DongoCollection):
    # if a specific database other than the default is desired, uncomment this:
    # database = 'myotherdatabase'
    collection = 'music_artists'
```

(continues on next page)

(continued from previous page)

```
# That is all you need to query and read records from the collection "music_artists",
# and the following would create new records and insert them and query for them.

queen = MusicArtist({
    'name': 'queen',
    'lead': 'freddie',
    'songs': ['we are the champions', 'we will rock you'],
    'fans': ['jack', 'jill'],
    'nested': {
        'field1': 1,
        'field2': 2,
    },
})
# insert must be called manually
queen.insert()

# you can use keywords and auto-insert with the "new" classmethod.
queen_stoneage = MusicArtist.new(
    name='queens of the stone age',
    lead='josh',
    songs=['go with the flow', 'little sister'],
    start=datetime(year=1996, month=1, day=1),
    fans=['jack'],
    nested={
        'field1': 1,
        'field2': 222,
    },
)

# queries are simple
for ma in MusicArtist.filter(fans='jack'):
    print('jack likes ' + ma['name'])

# you can even do regex queries and bulk updates
MusicArtist.filter(name__regex='^queen').update(new_field='this is a new field')

# There are many operators, like __gt, __gte, __lt, __lte, __in, __nin, all␣
↪corresponding to mongo's
# operators like $gt.

# you can do set logic as well with operators: |, &, ~
# for example less than comparisons and checking field existence:
for ma in (MusicArtist.filter(start__lt=datetime(2000, 1, 1)) | MusicArtist.
↪filter(start__exists=0)):
    print('either this music artist started before the year 2000 or their startdate␣
↪is unknown: ' + ma['name'])

# And you can query inside nested dictionaries

for ma in MusicArtist.filter(nested__field1=1):
    print(ma)

# updating the database or fetching fields is as easy as dictionary access
ma = MusicArtist.filter(name='queen').first()
ma['new_field'] = 'new_value'
print(ma['name'])
ma.set(new_field_2='a', new_field_3='b', new_field_4={'foo': 'bar'})
```

(continues on next page)

(continued from previous page)

```
ma['nested.field1'] = 'new value in nested field'
ma.set(nested__field1='reset that nested field to this value')
```

You will likely want methods associated with records, and to do that you just extend your class definition:

```
class Person(DongoCollection):
    collection = 'persons'

    def print_name(self):
        print(self.get('name', 'unknown'))

    def serialize(self):
        return {
            'name': self.get('name'),
            'age': self.get('age', 0),
            'birthday': self.get('start', datetime.min).isoformat(),
            'favorite_color': self.get('color'),
        }

    def change_color(self, new_color):
        # updates record in database as well
        self['color'] = new_color

    @classmethod
    def start_new_year(cls):
        # add 1 to all age values for every record with a field "age"
        cls.filter(age__exists=1).inc(age=1)
        # kill off those 110 and older
        cls.filter(age__gte=110).delete()

    @classmethod
    def startswith(cls, prefix):
        # find all persons with a name that starts with ``prefix``
        regex = '^{}'.format(prefix)
        return cls.filter(name__regex=regex)

    @classmethod
    def endswith(cls, suffix):
        # find all persons with a name that ends with ``suffix``
        regex = '{}$'.format(suffix)
        return cls.filter(name__regex=regex)

    @classmethod
    def first_10(cls):
        return cls.filter().iter(limit=10, sort='name')

    @classmethod
    def sort_by_oldest_first_then_alphabetically(cls):
        return cls.filter().iter(sort=[('age', -1), ('name', 1)])
```

CHAPTER 3

Release Notes

0.4.0 Added DongoBulk functionality, with lazy and bulk operations.

0.3.0 Added Dongo references feature, with instance.ref() and deref

0.2.3 Removed unnecessary dependency

0.2.2 Released alpha with python 2.7 and 3.x compatibility

0.2.1 Released alpha with python 3.x compatibility

CHAPTER 4

Dongo API

4.1 dongo

A Django-ORM inspired Mongo ODM.

dongo.**connect**(*database*, *host='localhost'*, *port=27017*, *uri=None*, *hosts=None*, *replica_set=None*)

Takes a default database, along with connection parameters. This also sets two global variables CLIENT and DATABASE_NAME. These are used for convenience for specifying a single connection which implicitly uses that client and database for all other defined classes. Otherwise, you can specify the database name in each class definition by adding the class variable `database = 'my_database_name'`. To not use a default database name, specify `connect(None)`

All you need to get started is to specify connect before defining `DongoCollection`'s:

```
# connects to localhost:27017
connect('mydatabase')

# connects to a remote host
connect('mydatabase', host='10.0.0.90')

# connects to a replica set
connect('mydatabase', hosts=['10.0.0.90', '10.0.0.91'],
        replica_set='a0')
```

Parameters

- **database** – the default database name, required
- **host** – the host string for single host connection, default localhost
- **hosts** – multiple hosts for replica sets (list of host or host:port)
- **replica_set** – the name of the replica set
- **port** – the mongo port, default 27017
- **uri** – optionally specify uri explicitly

Returns DongoClient instance

class dongo.**QuerySet**(*klass*, *query*)

This is the result from calling a `DongoCollection`'s `filter` classmethod.

You will only ever need to pull these by calling `filter`:

```
sel = MyDongoCollection.filter(some_field="value")
for item in sel:
    print(item)

sel.update(some_field="some new value")
```

To fetch all records with x=5 and foo='bar':

```
MyClass.filter(x=5, foo='bar')
```

To fetch all records where inside the "accounts" dictionary there's a "creditcard" property with value 1000:

```
MyClass.filter(accounts__creditcard=1000)
```

To fetch all records with x is greater or equal to 5:

```
MyClass.filter(x__gte=5)
```

To fetch all records where inside the "accounts" dictionary there's a "creditcard" property with value greater than 10000:

```
MyClass.filter(accounts__creditcard__gt=10000)
```

To fetch all records that have color property either red or blue:

```
MyClass.filter(color__in=['red', 'blue'])
```

To fetch all records where favorite color ISN'T blue:

```
MyClass.filter(color__ne='blue')
```

To fetch all records where favorite color isn't red or blue, where age is greater than 50, and where age is less than or equal to 60:

```
MyClass.filter(color__nin=['red', 'blue'], age__gt=50, age__lte=60)
```

count(***kwargs*)

Like `list` but performs a count.

Person.filter(color='red').count() # 10

Parameters

- **limit** – whether to limit records to a certain number
- ****kwargs** – other keywords to pass to pymongo

Returns the number of results it found

delete(***kwargs*)

Delete all documents that match the queryset:

```
# remove all accounts that haven't been active since 1/1/2000
Account.filter(last_active__lte=datetime(2000, 1, 1).delete()
```

Parameters ****kwargs** – keyword arguments to pass to pymongo's `delete_many`

Returns the pymongo delete_many result

first(***kwargs*)

Fetches the first matching filter or None.

Parameters

- **sort** – a field to sort results by initially, either a string or list of (string, direction) tuples like [('age', -1)]
- ****kwargs** – the keywords to pass to pymongo's `find_one`

Returns the class instance or None

first_or_die(***kwargs*)

Fetches the first matching filter or raises `DongoResultError`.

Parameters

- **sort** – a field to sort results by initially, either a string or list of (string, direction) tuples like [('age', -1)]
- ****kwargs** – the keywords to pass to pymongo's `find_one`

Returns the class instance

Raises `DongoResultError`

inc(***updates*)

Run database increments to all filtered items:

```
# Age each person by a year
Person.filter().update(age=1)
```

Parameters ****updates** – each field to increment and amount

Returns the pymongo update_many result

iter(*timeout=True*, ***kwargs*)

Performs a query from the keyword arguments. You can do the same by just iterating across the `.filter(...)` query, except `iter` allows you to pass keywords like `timeout`, `limit`, and `sort`.

Parameters

- **timeout** – whether the query should timeout, default `True`
- **sort** – a field to sort results by initially, either a string or list of (string, direction) tuples like [('age', -1)]
- **limit** – whether to limit records to a certain number
- ****kwargs** – other keywords to pass to pymongo

Yield each class instance from the query

list(*timeout=True*, ***kwargs*)

Returns a list of results rather than an iterator:

```
persons = Person.filter(age=21).list()
```

Parameters

- **timeout** – whether the query should timeout, default `True`
- **sort** – a field to sort results by initially, either a string or list of (string, direction) tuples like [('age', -1)]
- **limit** – whether to limit records to a certain number
- ****kwargs** – other keywords to pass to pymongo

Returns a list of the instances from the query results

map(*term*, ***kwargs*)

Takes a term and a set of keyword argument query and returns a dictionary keyed by the term values, pointing to a list of records where its term is that value.

For example, if there are three Person records and favorite color is red for two and blue for the last:

```
Person.filter(age__gt=25).map('color')
# {
#     'red': [Person(name=Joe, age=26), Person(name=Jack, age=30)],
#     'blue': [Person(name=Schmoe, age=42)],
# }
```

Parameters

- **timeout** – whether the query should timeout, default `True`
- **sort** – a field to sort results by initially, either a string or list of (string, direction) tuples like [('age', -1)]
- **limit** – whether to limit records to a certain number
- ****kwargs** – other keywords to pass to pymongo's `find`

Returns a dictionary of results `{term_value: [result1, ...]}`

update(***updates*)

Run database updates to all filtered items:

```
Person.filter(age=21).update(drinking=True)
```

Parameters ****updates** – each keyword update and value

Returns the pymongo update_many result

class dongo.**DongoCollection**(*data*)

This is the base class that all your collection classes will inherit from.

To start using dongo, you might connect then define a simple class:

```
from dongo import connect, DongoCollection

connect('mydatabase')

class User(DongoCollection):
```

(continues on next page)

(continued from previous page)

```
    collection = 'users'


for user in User.filter(username__regex='@example.org$'):
    print('user from example.org: ' + user['username'])

print('first 10')
for user in User.filter().iter(limit=10, sort='username'):
    print('user: {}'.format(user['username']))

new_user = User({'username': 'me@example.org'})
new_user.insert()

new_user2 = User.new(username='you@example.org')
# automatically inserted
```

classmethod bulk(*ops=None*)

Instanciate a fresh bulk operation to add bulk ops to. This will create an instance which you can invoke methods on it to update individual items, delete items, delete many, replace documents. However, it will be more efficient as it won't hit the database until you call `bulk.save()` which will run it as a single bulk operation.

Example:

```
persons = [person1, person2, person3, person4]

bulk = Person.bulk()
bulk.update_one(person1, age=31, name='jim')
bulk.update_one(person2, age=50)
bulk.inc_one(person3, age=1)
bulk.delete_one(person4)
bulk.delete_many(name='joe')
bulk.save()
```

Returns instance of DongoBulk

classmethod bulk_create(*objs*, *use_uuid=None*, ***kwargs*)

Insert many records into the database at once:

```
Person.bulk_create([
    {'name': 'joe', 'age': 21},
    {'name': 'bob', 'age': 22},
    ...
])
person1 = Person({'name': 'greg': 'age': 38})
person2 = Person({'name': 'bill': 'age': 45})
Person.bulk_create([person1, person2, ...])
```

Parameters **objs** – the dictionaries or class instances to insert in bulk

Returns the pymongo `insert_many` result

classmethod by_id(*_id*, ***kwargs*)

Find the record with the bson `ObjectId` value:

```
p = Person.by_id('6725b84b2401323bfda626e7')
```

Parameters **_id** – the `ObjectId` or string object id to check

Returns the class instance or None

classmethod by_ids(*ids*, ***kwargs*)
Find the records with the bson `ObjectId` values:

```
persons = Person.by_ids(['6725b84b2401323bfda626e7', ...])
```

Parameters **ids** – the `ObjectId` or string object id list to check

Returns a QuerySet of the records matching those IDs

classmethod by_uuid(*instance_uuid*, ***kwargs*)
Find the record with the uuid value:

```
p = Person.by_uuid('13bef77b-2a36-4d59-9339-42a5aa098833')
```

Parameters **instance_uuid** – the uuid to check

Returns the class instance or None

classmethod by_uuids(*uuids*, ***kwargs*)
Find many records from a list of uuids:

```
persons = Person.by_uuids([
    '13bef77b-2a36-4d59-9339-42a5aa098833',
    ...,
])
```

Parameters **uuids** – the uuids to check

Returns a QuerySet of the records matching those UUIDs

classmethod create_index(**args*, ***kwargs*)
Create an index on the collection, default in the background:

```
Person.create_index('name')
...
Person.create_index([('name', 1), ('age', -1)])
```

Parameters

- ***args** – the string term to index on, or a list of (term, direction) tuples
- **background** – whether to create the index in the background (default: True)

Returns the pymongo `create_index` result

delete(***kwargs*)
Delete the instance from the db:

```
p = Person.filter(name='joe').first()
p.delete()
p = Person.filter(name='joe').first()
print(p)
# None
```

Returns the pymongo `delete_one` result

classmethod filter(***query*)

Create a `QuerySet` based on the query:

```
for p in Person.filter(age__gte=21):
    print(p['name'] + ' can drink')
```

Parameters ****query** – the query to filter on

Returns the `QuerySet` instance

classmethod filter_and()

Creates a `QuerySet` with a top level $and, which you append to:

```
# Finds person where their age is 18 and name is 'Joe'
# Logically the same as Person.filter(name='Joe', age=18)
ms = Person.filter_and()
ms += Person.filter(name='Joe')
ms += Person.filter(age=18)
```

Returns the `QuerySet` instance

classmethod filter_or()

Creates a `QuerySet` with a top level $or, which you append to:

```
# Finds person where their age is 18 or 21
# Logically the same as Person.filter(age__in=[18, 21])
ms = Person.filter_or()
ms += Person.filter(age=21)
ms += Person.filter(age=18)
```

Returns the `QuerySet` instance

get(*field*, *default=None*)

Gets the value, or `default` Will work with . operator in key.

Example:

```
# foo is {"bar": {"baz": true}}
foo.get('bar.baz')
# returns True
```

Parameters

- **field** – the field to check
- **default** – default to return if it doesn't exist, default None

Returns the field value or `default`

classmethod get_one(***query*)
Performs a lookup to retrieve one record matching the query, like Django's `get`, as in `MyModel.objects.get(...)`. Raises exception if more than one instance is returned, like Django.

Parameters ****query** – the query to filter on

Returns the instance retrieved

Raises `DongoResultError`

inc(*field*, *amt=1*)
Performs an increment with an optional amount, default 1:

```
person = Person.filter().first()
person.inc('age')
person.inc('money', amt=100)
```

Parameters

- **field** – the field to increment
- **amt** – the amount to increment, default 1

Returns the result of the pymongo $inc `update`

insert(*use_uuid=None*, ***kwargs*)
Insert the class instance into the database:

```
p = Person({'name': 'joe', 'age': 21})
p.insert()
```

Returns the `ObjectId` of the new database record

json(*datetime_format=None*)
Serialize the object into JSON serializable data:

```
p = Person.new(name='joe', birthday=datetime(2000, 2, 1))
print(p.json())
# {'name': 'joe', 'birthday': '2000-02-01T00:00:00.000000'}
print(p.json(datetime_format='%d/%m/%Y')
# {'name': 'joe', 'birthday': '01/02/2000'}
```

Parameters **datetime_format** – the strftime datetime format string to use, or it just uses `isoformat()`

Returns the json serializable dictionary

lazy()
Returns an object that can have its fields be lazily updated through indexing, and subsequently saved:

```
p = Person.new(name='joe')
lazy = p.lazy()
lazy['age'] = 100
lazy['name'] = 'joejoe'
lazy.set(lastname='jimjim', birthday=datetime(1970, 1, 1))
lazy.save()
```

The attributes in the instance of Person are updated in Python memory, however the mongo updates aren't performed until save is called. If there's a possibility you caught an error with the save and ignored it, remember to perform an `instance.refresh_from_db()` to ensure you have what is reflected in the database.

Returns a `DongoLazyUpdater` wrapping the instance

classmethod map_by_ids(*ids*, ***kwargs*)

Find many records from a list of ids, and returns a map mapping passed id to record (or None if not found):

```
person_map = Person.map_by_ids([
    '6725b84b2401323bfda626e7',
    ...,
])
first_person = person_map['6725b84b2401323bfda626e7']
```

Parameters **uuids** – the uuids to check

Returns a dictionary mapping uuid => person or None

classmethod map_by_uuids(*uuids*, ***kwargs*)

Find many records from a list of uuids, and returns a map mapping passed uuid to record (or None if not found):

```
person_map = Person.by_uuids([
    '13bef77b-2a36-4d59-9339-42a5aa098833',
    ...,
])
first_person = person_map['13bef77b-2a36-4d59-9339-42a5aa098833']
```

Parameters **uuids** – the uuids to check

Returns a dictionary mapping uuid => person or None

classmethod new(***data*)

Instantly creates and inserts the new record.

Example:

```
person = Person.new({'name': 'Joe', 'age': 100})
```

ref()

Return a dongo reference to the record.

classmethod refresh_all_from_db(*instances*)

Updates all objects' data to most recently stored in Mongo. This is useful after bulk operations which don't update the instances' data themselves in Python, but have made db changes:

```
persons = Person.filter(age__gt=30).list()
bulk = Person.bulk()
for person in persons:
    bulk.update_one(person, older=True)
    bulk.inc_one(person, age=1)
Person.refresh_all_from_db(persons)
```

Returns None

refresh_from_db()
Updates object data to most recently stored in Mongo. If other processes might have updated it and you absolutely need the latest, refresh_from_db it:

```
person = Person.filter().first()
print(person)
import time ; time.sleep(30)
# see if it was updated by something else
person.refresh_from_db()
print(person)
```

Returns None

set(***kwargs*)
Performs a pymongo update using $set:

```
person = Person.filter().first()
person.set(name='new name', age=30)
```

Parameters ****kwargs** – each new field to update

Returns the pymongo `update` result

exception dongo.**DongoError**

exception dongo.**DongoConnectError**

exception dongo.**DongoResultError**

exception dongo.**DongoCollectionError**

class dongo.**DongoClient**(*host=None*, *port=None*, *document_class=<class 'dict'>*, *tz_aware=None*, *connect=None*, ***kwargs*)

exception dongo.**DongoDerefError**

dongo.**deref**(*data*)
Dereferences a dongo reference or collection of dongo references.

class dongo.**DongoBulk**(*klass*, *ops=None*)
A collection of bulk operations to be performed.

delete_many(***kwargs*)
Delete many documents.

person1 = Person.new(name='joe', age=30) person2 = Person.new(name='jill', age=31) person3 = Person.new(name='bob', age=50) bulk = Person.bulk() bulk.delete_many(age__gt=30) bulk.save() # Now jill and bob are gone, having age > 30

Parameters ****kwargs** – the query to run the delete many with

Returns the `pymongo.DeleteMany` result

delete_one(*instance*)
Delete a document.

person1 = Person.new(name='joe', age=30) person2 = Person.new(name='jill', age=40) bulk = Person.bulk() bulk.update_one(person, name='joejoe', age=50) bulk.delete_one(person2) bulk.save() Person.refresh_all_from_db([person1]) # Now jill is gone

Parameters **instance** – the instance to add a delete op to

Returns the `pymongo.DeleteOne` result

inc_one(*instance*, ***kwargs*)

Increment a single instance with an amount:

```
person1 = Person.new(name='joe', age=30)
person2 = Person.new(name='jill', age=40)
bulk = Person.bulk()
bulk.inc_one(person, age=1)
bulk.inc_one(person2, age=1)
bulk.save()
Person.refresh_all_from_db([person1, person2])
# Now joe and jill are 1 year older
```

Parameters **instance** – the instance to add an increment op to

Returns the `pymongo.UpdateOne` result

replace_one(*instance*, ***kwargs*)

Replace a single instance's document entirely:

```
person1 = Person.new(name='joe', age=30)
person2 = Person.new(name='jill', age=40)
bulk = Person.bulk()
bulk.replace_one(person, name='joejoe', age=50)
bulk.replace_one(person2, name='jilly', age=60)
bulk.save()
Person.refresh_all_from_db([person1, person2])
# Now their documents are overwritten entirely
```

Parameters **instance** – the instance to add a replace op to

Returns the `pymongo.ReplaceOne` result

save()

Perform all the saved bulk operations.

Returns the result from the `pymongo.bulk_write`

take(*lazy*)

Take the bulk operations that a DongoLazyUpdater or other DongoBulk was going to perform to aggregate into one DongoBulk:

```
person1 = Person.new(name='joe')
person2 = Person.new(name='jill')
lazy1 = person1.lazy()
lazy2 = person2.lazy()
lazy1.set(name='joejoe', age=30)
lazy2.set(age=50)
lazy1['foo'] = 'bar'

bulk = Person.bulk()
bulk.take(lazy1)
bulk.take(lazy2)

bulk2 = Person.bulk()
```

(continues on next page)

(continued from previous page)

```
bulk2.update_one(person1, name='hello')
bulk.take(bulk2)

bulk.save()

lazy1['favorite_number'] = 12
# this only performs the one ``favorite_number`` update
lazy1.save()
```

The shorthand for take is simply addition assignment:

```
bulk += lazy1 + bulk2 + lazy2 + bulk3
bulk.save()
```

You can simply add DongoLazyUpdaters together as well:

```
(lazy1 + lazy2).save()
```

Parameters **lazy** – the `DongoLazyUpdater` or other `DongoBulk`

update_one(*instance*, ***kwargs*)

Update a single instance with a deferred update operation:

```
person1 = Person.new(name='joe', age=30)
person2 = Person.new(name='jill', age=40)
bulk = Person.bulk()
bulk.update_one(person, name='joejoe')
bulk.update_one(person2, name='jilljill', age=41)
bulk.save()
Person.refresh_all_from_db([person1, person2])
# joe is named joejoe, and jill is named jilljill and 41 years old
```

Parameters **instance** – the instance to add an update op to

Returns the `pymongo.UpdateOne` result

class dongo.**DongoLazyUpdater**(*instance*)

A wrapper around a DongoBulk and DongoCollection instance to allow lazy updates.

save()

Perform the deferred updates in a bulk operation.

Returns the result of `pymongo.bulk_write`

set(***kwargs*)

Assign some deferred updates to be performed on `save()`

Parameters ****kwargs** – the fields to update with their value

Returns the result of `pymongo.UpdateOne`

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

T

U