MongoTor Documentation

Release 0.1.0

Marcel Nicolat

June 11, 2014

Contents

(MONGO + TORnado) is an asynchronous toolkit for accessing mongo with tornado.

CHAPTER 1

Features

- ORM like to map documents and fields `orm`
- Advanced connection management (replica sets, slave okay)
- Automatic reconnection
- Connection pooling
- Support for running database commands (find, find_one, count, sum, mapreduce etc...)
- Signals for pre_save, post_save, pre_remove, post_remove, pre_update and post_update
- 100% of code coverage by test

CHAPTER 2

Contents:

2.1 Installation

2.1.1 Supported Installation Methods

MongoTor supports installation using standard Python "distutils" or "setuptools" methodologies. An overview of potential setups is as follows:

2.1.2 Install via easy_install or pip

When `easy_install` or `pip` is available, the distribution can be downloaded from Pypi and installed in one step:

```
easy_install mongotor
```

Or with pip:

```
pip install mongotor
```

This command will download the latest version of MongoTor from the Python Cheese Shop and install it to your system.

2.1.3 Installing using setup.py

Otherwise, you can install from the distribution using the `setup.py` script:

```
python setup.py install
```

2.1.4 Checking the Installed MongoTor Version

The version of MongoTor installed can be checked from your Python prompt like this:

```
>>> import mongotor
>>> mongotor.version
```

2.1.5 Requirements

The following three python libraries are required.

- pymongo version 1.9+ for bson library
- tornado

Note: The above requirements are automatically managed when installed using any of the supported installation methods

2.2 Tutorial

This tutorial is meant to introduce you to the basic concepts of using MongoTor using an example application. The example application is a simple user database where people could fill in their information and register themselves.

2.2.1 Getting started

- Ensure that an instance of MongoDB is running in an accessible location. This tutorial assumes that such an instance is running on the localhost.

2.2.2 Defining our Collection

A MongoDB Collection is the rough equivalent of a table in a relational database. Though MongoDB collections are schemaless documents in them usually have a similar structure. This "similar structure" could be defined as a `Collection`.

In this example application we define the structure of Users collection with the required `field` (s)

```
class User(Collection):
    __collection__ = "user"

    _id = ObjectIdField()
    name = StringField()
    active = BooleanField()
    created = DateTimeField()
```

2.2.3 Connecting to the Database

A connection to the MongoDB database needs to be established before MongoTor can manage collections or do any other operations. A connection is established using a `Database` object

```
from mongotor.database import Database
Database.connect('localhost:27017', 'test_db')
```

2.2.4 Creating a new document

A new document can be created in the collection by creating an instance of the Collection, assigning values to the fields and then calling the save method

```
new_user = User()
new_user.name = "New user"
new_user.active = True
new_user.save()
```

A new instance would also be created from a dictionary (for example from a Form handler in your web application):

```
>>> new_user = User.create({'name': 'Some user name'})
>>> new_user.name
u'Some user name'
>>> new_user.save()
```

2.2.5 Using ORM in a TornadoHandler

```
from mongotor.orm import Collection
from mongotor.orm.field import StringField, ObjectIdField, BooleanField, DateTimeField
from mongotor.database import Database

from datetime import datetime
import tornado.web
from tornado import gen

# A connection to the MongoDB database needs to be established before perform operations
# A connection is stabilished using a Databse object
Database.connect(['localhost:27017'], 'asyncmongo_test')

class User(Collection):

    __collection__ = "user"

    _id = ObjectIdField()
    name = StringField()
    active = BooleanField()
    created = DateTimeField()

class Handler(tornado.web.RequestHandler):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        user = User()
        user.name = "User name"
        user.active = True
        user.created = datetime.now()

        yield gen.Task(user.save)

        # update date
        user.name = "New name"
        yield gen.Task(user.update)

        # find one object
        user_found = yield gen.Task(User.objects.find_one, user._id)

        # find many objects
        new_user = User()
        new_user.name = "new user name"
```

```
        new_user.user.active = True
        new_user.created = datetime.now()

        users_actives = yield gen.Task(User.objects.find, {'active': True})

        users_actives[0].active = False
        yield gen.Task(users_actives[0].save)

        # remove object
        yield gen.Task(user_found.remove)
```

2.2.6 Using Client in a TornadoHandler

MongoTor supports `Client` for direct access to mongo, without orm layer

```
from mongotor.database import Database
from bson import ObjectId
from tornado import gen, web

class Handler(web.RequestHandler):

    def initialize(self):
        self.db = Database.connect(['localhost:27017'], 'asyncmongo_test')

    @web.asynchronous
    @gen.engine
    def get(self):
        user = {'_id': ObjectId, 'name': 'User Name'}
        yield gen.Task(self.db.user.insert, user)

        yield gen.Task(self.db.user.update, user['_id'], {"$set": {'name': 'New User Name'}})

        user_found = yield gen.Task(self.db.user.find_one, user['_id'])
        assert user_found['name'] == 'New User Name'

        yield gen.Task(self.db.user.remove, user['_id'])
```

2.2.7 Using Signals

MongoTor supports `signals` for pre_save, post_save, pre_remove, post_remove, pre_update, post_update to which receivers could bind to.

```
from mongotor.orm import collection, field
from mongotor.orm.signal import pre_save, receiver
from mongotor.database import Databas
from bson import ObjectId

import tornado.web
from tornado import gen

class User(collection.Collection):
    __collection__ = "user"

    _id = field.ObjectIdField()
    name = field.StringField()
```

```
    active = field.BooleanField()
    created = field.DateTimeField()

@receiver(pre_save, User)
def set_object_id(sender, instance):
    if not instance._id:
        instance._id = ObjectId()


class Handler(tornado.web.RequestHandler):

    @tornado.web.asynchronous
    @gen.engine
    def get(self):
        user = User()
        user.name = "User name"
        user.active = True
        user.created = datetime.now()

        yield gen.Task(user.save)
```

2.3 API Reference

`mongotor.`**`version`** = **'0.1.0'**

str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

`mongotor.`**`Database`**

Alias for `mongotor.database.Database`.

Sub-modules:

2.3.1 `database` – Database level operations

class `mongotor.database.`**`Database`**

Database object

classmethod **`connect`**(**args*, ***kwargs*)

connect database

this method is deprecated, use `init` to initiate a new database

classmethod **`disconnect`**()

Disconnect to database

```
>>> Database.disconnect()
```

`command`(**args*, ***kwargs*)

Issue a MongoDB command.

Send command *command* to the database and return the response. If *command* is an instance of `basestring` then the command {*command*: *value*} will be sent. Otherwise, *command* must be an instance of `dict` and will be sent as is.

Any additional keyword arguments will be added to the final command document before it is sent.

For example, a command like `{buildinfo: 1}` can be sent using:

```
>>> db.command("buildinfo")
```

For a command where the value matters, like `{collstats: collection_name}` we can do:

```
>>> db.command("collstats", collection_name)
```

For commands that take additional arguments we can use kwargs. So `{filemd5: object_id, root: file_root}` becomes:

```
>>> db.command("filemd5", object_id, root=file_root)
```

Parameters

- *command*: document representing the command to be issued, or the name of the command (for simple commands only).

  Note: the order of keys in the *command* document is significant (the "verb" must come first), so commands which require multiple keys (e.g. *findandmodify*) should use an instance of `SON` or a string and kwargs instead of a Python *dict*.

- *value* (optional): value to use for the command verb when *command* is passed as a string
- ***kwargs* (optional): additional keyword arguments will be added to the command document before it is sent

2.3.2 `collection` – A mongo collection

class `mongotor.client.`**`Client`**(*database*, *collection*)

`insert`(**args*, ***kwargs*)

Insert a document

Parameters

- *doc_or_docs*: a document or list of documents to be inserted
- *safe* (optional): check that the insert succeeded?
- *check_keys* (optional): check if keys start with '$' or contain '.', raising `InvalidName` in either case
- *callback* : method which will be called when save is finished

`remove`(**args*, ***kwargs*)

remove a document

Parameters

- *spec_or_id*: a query or a document id
- *safe* (optional): safe insert operation
- *callback* : method which will be called when save is finished

`update`(**args*, ***kwargs*)

Update a document(s) in this collection.

Parameters

- *spec*: a `dict` or `SON` instance specifying elements which must be present for a document to be updated
- *document*: a `dict` or `SON` instance specifying the document to be used for the update or (in the case of an upsert) insert - see docs on MongoDB **'update modifiers'_**
- *upsert* (optional): perform an upsert if `True`
- *safe* (optional): check that the update succeeded?
- *multi* (optional): update all documents that match *spec*, rather than just the first matching document. The default value for *multi* is currently `False`, but this might eventually change to `True`. It is recommended that you specify this argument explicitly for all update operations in order to prepare your code for that change.

find_one(**args*, ***kwargs*)

Get a single document from the database.

All arguments to `find()` are also valid arguments for `find_one()`, although any *limit* argument will be ignored. Returns a single document, or `None` if no matching document is found.

Parameters

- *spec_or_id* (optional): a dictionary specifying the query to be performed OR any other type to be used as the value for a query for `"_id"`.
- ***kwargs* (optional): any additional keyword arguments are the same as the arguments to `find()`.

find(**args*, ***kwargs*)

Query the database.

The *spec* argument is a prototype document that all results must match. For example:

Parameters

- *spec* (optional): a SON object specifying elements which must be present for a document to be included in the result set
- *fields* (optional): a list of field names that should be returned in the result set ("_id" will always be included), or a dict specifying the fields to return
- *skip* (optional): the number of documents to omit (from the start of the result set) when returning the results
- *limit* (optional): the maximum number of results to return
- *timeout* (optional): if True, any returned cursor will be subject to the normal timeout behavior of the mongod process. Otherwise, the returned cursor will never timeout at the server. Care should be taken to ensure that cursors with timeout turned off are properly closed.
- *snapshot* (optional): if True, snapshot mode will be used for this query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution. For details, see the snapshot documentation.
- *tailable* (optional): the result of this find call will be a tailable cursor - tailable cursors aren't closed when the last data is retrieved but are kept open and the cursors location marks the final document's position. if more data is received iteration of the cursor will continue from the last document received. For details, see the tailable cursor documentation.

- *sort* (optional): a list of (key, direction) pairs specifying the sort order for this query. See `sort()` for details.
- *max_scan* (optional): limit the number of documents examined when performing the query
- *read_preferences* (optional): The read preference for this query.

count(*callback*)

Get the size of the results among all documents.

Returns the number of documents in the results set

distinct(*key*, *callback*)

Get a list of distinct values for *key* among all documents in this collection.

Raises `TypeError` if *key* is not an instance of `basestring` (`str` in python 3).

To get the distinct values for a key in the result set of a query use `distinct()`.

Parameters

- *key*: name of key for which we want to get the distinct values

aggregate(**args*, ***kwargs*)

Perform an aggregation using the aggregation framework on this collection.

Parameters

- *pipeline*: a single command or list of aggregation commands
- *read_preference*

Note: Requires server version **>= 2.1.0**

group(**args*, ***kwargs*)

Perform a query similar to an SQL *group by* operation.

Returns an array of grouped items.

The *key* parameter can be:

- `None` to use the entire document as a key.
- A `list` of keys (each a `basestring` (`str` in python 3)) to group by.
- A `basestring` (`str` in python 3), or `Code` instance containing a JavaScript function to be applied to each document, returning the key to group by.

Parameters

- *key*: fields to group by (see above description)
- *condition*: specification of rows to be considered (as a `find()` query specification)
- *initial*: initial value of the aggregation counter object
- *reduce*: aggregation function as a JavaScript string
- *finalize*: function to be called on each object in output list.

2.3.3 `orm` – Map a mongo collection into a python class

class `mongotor.orm.collection.`**`Collection`**

Collection is the base class

This class map a mongo collection into a python class. You only need to write a class and starts to use the orm advantages.

For example, a simple users collection can be mapping using:

```
>>> from mongotor.orm import collection, field
>>> class Users(collection.Collection):
>>>     __collection__ = 'users'
>>>     name = field.StringField()
```

save(**args*, ***kwargs*)

Save a document

```
>>> user = Users()
>>> user.name = 'should be name'
>>> user.save()
```

Parameters

- *safe* (optional): safe insert operation
- *check_keys* (optional): check if keys start with '$' or contain '.', raising `InvalidName` in either case

•*callback* : method which will be called when save is finished

remove(**args*, ***kwargs*)

Remove a document

Parameters

•*safe* (optional): safe remove operation

•*callback* : method which will be called when remove is finished

update(**args*, ***kwargs*)

Update a document

Parameters

•*safe* (optional): safe update operation

•*callback* : method which will be called when update is finished

•*force*: if True will overide full document

2.3.4 `errors` – Mongotor errors

class `mongotor.errors.`**`Error`**

Base class for all mongotor exceptions.

class `mongotor.errors.`**`InterfaceError`**

Raised when a connection to the database cannot be made or is lost.

class `mongotor.errors.`**`TooManyConnections`**

Raised when a pool is busy.

class `mongotor.errors.`**`InvalidOperationError`**

Raised when a client attempts to perform an invalid operation.

class `mongotor.errors.`**`IntegrityError`**(*msg*, *code=None*)

Raised when a safe insert or update fails due to a duplicate key error.

CHAPTER 3

Contributing to the project

List of contributors

3.1 Source Code

The source is available on GitHub and contributions are welcome.

3.2 Issues

Please report any issues via github issues

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

m