FREE eBook

LEARNING PyMongo

Free unaffiliated eBook created from
Stack Overflow contributors.

#pymongo

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pymongo

It is an unofficial and free PyMongo ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PyMongo.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with PyMongo

Remarks

This section provides an overview of what pymongo is, and why a developer might want to use it.

It should also mention any large subjects within pymongo, and link out to the related topics. Since the Documentation for pymongo is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting pymongo set up or installed.

- Installing with Pip
    - To install pymongo for the first time:

      ```
      pip install pymongo
      ```

    - Installing a specific version of pymongo:

      Where X.X.X is the version to be installed

      ```
      pip install pymongo==X.X.X
      ```

    - Upgrading existing pymongo:

      ```
      pip install --upgrade pymongo
      ```

- Installing with easy_install
    - To install pymongo for the first time:

      ```
      python -m easy_install pymongo
      ```

    - Upgrading existing pymongo:

      ```
      python -m easy_install -U pymongo
      ```

Hello, World

PyMongo is a native Python driver for MongoDB.

Install PyMongo

```
pip install pymongo
```

Create a connection

Use MongoClient to create a connection. MongoClient defaults to MongoDB instance running on `localhost:27017` if not specified.

```
from pymongo import MongoClient
client = MongoClient()
```

Access Database Objects

PyMongo's Database class represents database construct in MongoDB. Databases hold groups of logically related collections.

```
db = client.mydb
```

Access Collection Objects

PyMongo's Collection class represents collection construct in MongoDB. Collections hold groups of related documents.

```
col = db.mycollection
```

MongoDB creates new databases and collections implicitly upon first use.

Basic CRUD Operation

MongoDB stores data records as BSON *documents*. BSON is the binary representation of JSON.

```
$ python
>>> from pymongo import MongoClient
>>> client = MongoClient()
>>> col = client.mydb.test
```

Create

Insert a single document `insert_one(document)`

```
>>> result = col.insert_one({'x':1})
>>> result.inserted_id
ObjectId('583c16b9dc32d44b6e93cd9b')
```

Insert multiple documents `insert_many(documents)`

```
>>> result = col.insert_many([{'x': 2}, {'x': 3}])
>>> result.inserted_ids
[ObjectId('583c17e7dc32d44b6e93cd9c'), ObjectId('583c17e7dc32d44b6e93cd9d')]
```

Replace a single document matching the filter `replace_one(filter, replacement, upsert=False)`. (to insert a new document if matching document doesn't exist, use `upsert=True`)

```
>>> result = col.replace_one({'x': 1}, {'y': 1})
>>> result.matched_count
1
>>> result.modified_count
1
```

Update

Update a single document matching the filter `update_one(filter, update, upsert=False)`

```
>>> result = col.update_one({'x': 1}, {'x': 3})
```

Update one or more documents that match the filter `update_many(filter, update, upsert=False)`

```
>>> result = col.update_many({'x': 1}, {'x': 3})
```

Read

Query the database `find(filter=None, projection=None, skip=0, limit=0, no_cursor_timeout=False)`. The *filter* argument is a prototype document that all results must match.

```
>>> result = col.find({'x': 1})
```

Get a single document from the database `find_one(filter=None)`

```
>>> result = col.find_one()
```

Query With Projection

```
query={'x':1}
projection={'_id':0, 'x':1} # show x but not show _id
result=col.find(query,projection)
```

Delete

Delete a single document matching the filter `delete_one(filter)`

```
>>> result = col.delete_one({'x': 1})
>>> result.deleted_count
```

```
1
```

Delete one or more documents matching the filter `delete_many(filter)`

```
>>> result = col.delete_many({'x': 1})
>>> result.deleted_count
3
```

PyMongo also provides `find_one_and_delete()`, `find_one_and_update()` and `find_one_and_replace()` functionality.

Read Getting started with PyMongo online: https://riptutorial.com/pymongo/topic/2612/getting-started-with-pymongo

Chapter 2: Converting between BSON and JSON

Introduction

In many applications, records from MongoDB need to be serialized in JSON format. If your records have fields of type date, datetime, objectId, binary, code, etc. you will encounter `TypeError: not JSON serializable` exceptions when using `json.dumps`. This topic shows how to overcome this.

Examples

Using json_util

json_util provides two helper methods, `dumps` and `loads`, that wrap the native json methods and provide explicit BSON conversion to and from json.

Simple usage

```
from bson.json_util import loads, dumps
record = db.movies.find_one()
json_str = dumps(record)
record2 = loads(json_str)
```

if `record` is:

```
{
    "_id" : ObjectId("5692a15524de1e0ce2dfcfa3"),
    "title" : "Toy Story 4",
    "released" : ISODate("2010-06-18T04:00:00Z")
}
```

then `json_str` becomes:

```
{
    "_id": {"$oid": "5692a15524de1e0ce2dfcfa3"},
    "title" : "Toy Story 4",
    "released": {"$date": 1276833600000}
}
```

JSONOptions

It is possible to customize the behavior of `dumps` via a `JSONOptions` object. Two sets of options are already available: `DEFAULT_JSON_OPTIONS` and `STRICT_JSON_OPTIONS`.

```
>>> bson.json_util.DEFAULT_JSON_OPTIONS
```

```
JSONOptions(strict_number_long=False, datetime_representation=0,
 strict_uuid=False, document_class=dict, tz_aware=True,
 uuid_representation=PYTHON_LEGACY, unicode_decode_error_handler='strict',
 tzinfo=<bson.tz_util.FixedOffset object at 0x7fc168a773d0>)
```

To use different options, you can:

1. modify the `DEFAULT_JSON_OPTIONS` object. In this case, the options will be used for all subsequent call to `dumps`:

```
from bson.json_util import DEFAULT_JSON_OPTIONS
DEFAULT_JSON_OPTIONS.datetime_representation = 2
dumps(record)
```

2. specify a `JSONOptions` in a call to `dumps` using the `json_options` parameter:

```
# using strict
dumps(record, json_options=bson.json_util.STRICT_JSON_OPTIONS)

# using a custom set of options
from bson.json_util import JSONOptions
options = JSONOptions() # options is a copy of DEFAULT_JSON_OPTIONS
options.datetime_representation=2
dumps(record, json_options=options)
```

The parameters of `JSONOptions` are:

- **strict_number_long**: If true, Int64 objects are encoded to MongoDB Extended JSON's Strict mode type NumberLong, ie `{"$numberLong": "<number>" }`. Otherwise they will be encoded as an int. Defaults to False.
- **datetime_representation**: The representation to use when encoding instances of datetime.datetime. 0 => `{"$date": <dateAsMilliseconds>}`, 1 => `{"$date": {"$numberLong": "<dateAsMilliseconds>"}}`, 2 => `{"$date": "<ISO-8601>"}`
- **strict_uuid**: If true, uuid.UUID object are encoded to MongoDB Extended JSON's Strict mode type Binary. Otherwise it will be encoded as `{"$uuid": "<hex>" }`. Defaults to False.
- **document_class**: BSON documents returned by loads() will be decoded to an instance of this class. Must be a subclass of collections.MutableMapping. Defaults to dict.
- **uuid_representation**: The BSON representation to use when encoding and decoding instances of uuid.UUID. Defaults to PYTHON_LEGACY.
- **tz_aware**: If true, MongoDB Extended JSON's Strict mode type Date will be decoded to timezone aware instances of datetime.datetime. Otherwise they will be naive. Defaults to True.
- **tzinfo**: A `datetime.tzinfo` subclass that specifies the timezone from which datetime objects should be decoded. Defaults to utc.

Using python-bsonjs

python-bsonjs does not depend on PyMongo and can offer a nice performance improvement over `json_util`:

bsonjs is roughly 10-15x faster than PyMongo's json_util at decoding BSON to JSON and encoding JSON to BSON.

Note that:

- to use bsonjs effectively, it is recommended to work directly with `RawBSONDocument`
- dates are encoded using the LEGACY representation, i.e. `{"$date": <dateAsMilliseconds>}`. There is currently no options to change that.

Installation

```
pip install python-bsonjs
```

Usage

To take full advantage of the bsonjs, configure the database to use the `RawBSONDocument` class. Then, use `dumps` to convert bson raw bytes to json and `loads` to convert json to bson raw bytes:

```
import pymongo
import bsonjs
from pymongo import MongoClient
from bson.raw_bson import RawBSONDocument

# configure mongo to use the RawBSONDocument representation
db = pymongo.MongoClient(document_class=RawBSONDocument).samples
# convert json to a bson record
json_record = '{"_id": "some id", "title": "Awesome Movie"}'
raw_bson = bsonjs.loads(json_record)
bson_record = RawBSONDocument(raw_bson)
# insert the record
result = db.movies.insert_one(bson_record)
print(result.acknowledged)

# find some record
bson_record2 = db.movies.find_one()
# convert the record to json
json_record2 = bsonjs.dumps(bson_record2.raw)
print(json_record2)
```

Using the json module with custom handlers

If all you need is serializing mongo results into json, it is possible to use the `json` module, provided you define custom handlers to deal with non-serializable fields types. One advantage is that you have full power on how you encode specific fields, like the datetime representation.

Here is a handler which encodes dates using the iso representation and the id as an hexadecimal string:

```
import pymongo
import json
import datetime
```

```
import bson.objectid

def my_handler(x):
    if isinstance(x, datetime.datetime):
        return x.isoformat()
    elif isinstance(x, bson.objectid.ObjectId):
        return str(x)
    else:
        raise TypeError(x)

db = pymongo.MongoClient().samples
record = db.movies.find_one()
# {u'_id': ObjectId('5692a15524de1e0ce2dfcfa3'), u'title': u'Toy Story 4',
#   u'released': datetime.datetime(2010, 6, 18, 4, 0),}

json_record = json.dumps(record, default=my_handler)
# '{"_id": "5692a15524de1e0ce2dfcfa3", "title": "Toy Story 4",
#    "released": "2010-06-18T04:00:00"}'
```

Read Converting between BSON and JSON online:
https://riptutorial.com/pymongo/topic/9348/converting-between-bson-and-json

Chapter 3: Filter documents by creation time stored in ObjectId

Introduction

Includes pymongo query examples to filter documents by timestamp encapsulated in ObjectId

Examples

Documents created in the last 60 seconds

How to find documents created 60 seconds ago

```
seconds = 60

gen_time = datetime.datetime.today() - datetime.timedelta(seconds=seconds)
dummy_id = ObjectId.from_datetime(gen_time)

db.CollectionName.find({"_id": {"$gte": dummy_id}})
```

If you're in a different timezone, you may need to offset the datetime to UTC

```
seconds = 60

gen_time = datetime.datetime.today() - datetime.timedelta(seconds=seconds)
# converts datetime to UTC
gen_time=datetime.datetime.utcfromtimestamp(gen_time.timestamp())

dummy_id = ObjectId.from_datetime(gen_time)

db.Collection.find({"_id": {"$gte": dummy_id}})
```

Read Filter documents by creation time stored in ObjectId online:
https://riptutorial.com/pymongo/topic/9855/filter-documents-by-creation-time-stored-in-objectid

Credits

S. No	Chapters	Contributors
1	Getting started with PyMongo	Community, Himavanth, Kheshav Sewnundun, tim
2	Converting between BSON and JSON	Derlin
3	Filter documents by creation time stored in ObjectId	Sawan Vaidya