MongoEngine GoodJSON Documentation

Hiroaki Yamamoto

Jan 02, 2023

Contents:

1 Getting Started **3**
- 1.1 Why MongoEngine GoodJSON created . . . 3
- 1.2 Installation . . . 4

2 Basic Use **7**
- 2.1 Document Inheritance . . . 7
- 2.2 Follow Reference . . . 8

3 Addtional Features **11**
- 3.1 FollowReferenceField . . . 11
- 3.2 Exclude fields from JSON serialization/deserialization . . . 11
- 3.3 Reference Limit . . . 12

4 Dive in the deep **15**
- 4.1 Encoder / Decoder . . . 15
- 4.2 FAQ from issue tracker . . . 15

5 MIT License **17**

6 Contribution **19**
- 6.1 Posting Issues . . . 19
- 6.2 Solve your problem by your hand . . . 19
- 6.3 Testbed Environment . . . 19
- 6.4 Pull Request . . . 20

7 Indices and tables **21**

This document describes how to use MongoEngine GoodJSON and FAQs that was asked on issues tracker

CHAPTER 1

Getting Started

1.1 Why MongoEngine GoodJSON created

1.1.1 Problem

Using MongoEngine to create something (e.g. RESTful API), sometimes you might want to serialize the data from the db into JSON, but some fields are weird and not suitable for frontend/api:

```
{
  "_id": {
    "$oid": "5700c32a1cbd5856815051ce"
  },
  "name": "Hiroaki Yamamoto",
  "registered_date": {
      "$date": 1459667811724
  }
}
```

If you don't mind about `_id`, `$oid`, and `$date`, it's fine. However, these data might cause problems when you using AngularJS, because prefix `$` is reserved by the library.

In addition to this, object in object might cause `No such property $oid of undefined` error when you handle the data like above on the frontend.

1.1.2 The Solution

To solve the problems, the generated data should be like this:

```
{
  "id": "5700c32a1cbd5856815051ce",
  "name": "Hiroaki Yamamoto",
  "registered_date": 1459667811724
}
```

Making above structure can be possible by doing re-mapping, but if we do it on API's controller object, the code might get super-dirty:

```
"""Dirty code."""
import mongoengine as db


class User(db.Document):
  """User class."""
  name = db.StringField(required=True, unique=True)
  registered_date = db.DateTimeField()


def get_user(self):
  """Get user."""
  models = [
    {
      ("id" if key == "_id" else key): (
        value.pop("$oid") if "$oid" in value and isinstance(value, dict)
        else value.pop("$date") if "$date" in value and isinstance(value, dict)
        else value  #What if there are the special fields in child dict?
      )
      for (key, value) in doc.items()
    } for doc in User.objects(pk=ObjectId("5700c32a1cbd5856815051ce"))
  ]
  return json.dumps(models, indent=2)
```

To give the solution of this problem, I developed this scirpt. By using this script, you will not need to make the transform like above. i.e.

```
"""A little-bit clean code."""

import mongoengine as db
import mongoengine_goodjson as gj


class User(gj.Document):
  """User class."""
  name = db.StringField(required=True, unique=True)
  registered_date = db.DateTimeField()


def get_user(self):
  """Get user."""
  return model_cls.objects(
    pk=ObjectId("5700c32a1cbd5856815051ce")
  ).to_json(indent=2)
```

1.2 Installation

There's several ways to install MongoEngine GoodJSON. The easiest way is to install thru pypi

```
pip install mongoengine_goodjson
```

As an alternative way, you can download the code from github release, extract the tgz archive, and execute setup.py:

```
python setup.py install
```

However, if you are able to create virtual environment, you can create one **before installing this script**.:

```
python -m venv venv
```

CHAPTER 2

Basic Use

2.1 Document Inheritance

First of all, let's see the usual ODM:

```
import mongoengine as db


class Address(db.EmbeddedDocument):
  """Address schema."""

  street = db.StringField()
  city = db.StringField()
  state = db.StringField()


class User(db.Document):
  """User data schema."""

  name = db.StringField()
  email = db.EmailField()
  address = db.EmbeddedDocumentListField(Address)
```

As you can see the code, this code has nothing special. And, when you serialize the instance into JSON, you will get:

```
{
  "id": { "$oid": "5700c32a1cbd5856815051ce" },
  "name": "Example Man",
  "email": "test@example.com",
  "address": [
    {
      "street": "Hello Street",
      "city": "Hello City",
      "state": "Hello State"
```

(continues on next page)

(continued from previous page)

```
    }, {
      "street": "World Street",
      "city": "World City",
      "state": "World State"
    }
  ]
}
```

And yes, we want to replace `$oid` object with `str` that shows `5700c32a1cbd5856815051ce`. MongoEngine enables to you do it very easily. Let's just inherit `mongoengine_goodjson.Document` like this:

```
import mongoengine_goodjson as gj
import mongoengine as db


class Address(gj.EmbeddedDocument):
    """Address schema."""

    street = db.StringField()
    city = db.StringField()
    state = db.StringField()


class User(gj.Document):
    """User data schema."""

    name = db.StringField()
    email = db.EmailField()
    address = db.EmbeddedDocumentListField(Address)
```

Then, running `user.to_json` (`user` is the instance object of User), you will get the JSON code like this:

```
{
  "id": "5700c32a1cbd5856815051ce",
  "name": "Example Man",
  "email": "test@example.com",
  "address": [
    {
      "street": "Hello Street",
      "city": "Hello City",
      "state": "Hello State"
    }, {
      "street": "World Street",
      "city": "World City",
      "state": "World State"
    }
  ]
}
```

2.2 Follow Reference

Let's see ODM using `ReferenceField`.

```python
import mongoengine as db
import mongoengine_goodjson as gj


class Book(gj.Document):
  """Book information model."""

  name = db.StringField(required=True)
  isbn = db.StringField(required=True)
  author = db.StringField(required=True)
  publisher = db.StringField(required=True)
  publish_date = db.DateTimeField(required=True)


class User(gj.Document):
  firstname = db.StringField(required=True)
  lastname = db.StringField(required=True)
  books_bought = db.ListField(db.ReferenceField(Book))
  favorite_one = db.ReferenceField(Book)
```

And here is the JSON data:

```json
{
  "id": "570ee9d1fec55e755db82129",
  "firstname": "James",
  "lastname": "Smith",
  "books_bought": [
    "570eea0afec55e755db8212a",
    "570eea0bfec55e755db8212b",
    "570eea0bfec55e755db8212c"
  ],
  "favorite_one": "570eea0bfec55e755db8212b"
}
```

This seems to be good deal for `Reference Field`, but sometimes you might want to generate the Document with Referenced Document like Embedded Document like this:

```json
{
  "id": "570ee9d1fec55e755db82129",
  "firstname": "James",
  "lastname": "Smith",
  "books_bought": [
    {
      "id": "570eea0afec55e755db8212a",
      "name": " ()",
      "author": " ",
      "publisher": "",
      "publish_date": "1976-10-01",
      "isbn": "978-4041366035"
    },
    {
      "id": "570eea0bfec55e755db8212b",
      "name": " ()",
      "author": " ",
      "publisher": "",
      "publish_date": "1976-10-01",
      "isbn": "978-4041366042"
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "id": "570eea0bfec55e755db8212c",
      "name": "The Voynich Manuscript: Full Color Photographic Edition",
      "author": "Unknown",
      "publisher": "FQ Publishing",
      "publish_date": "2015-01-17",
      "isbn": "978-1599865553"
    }
  ],
  "favorite_one": {
    "id": "570eea0bfec55e755db8212b",
    "name": " ()",
    "author": " ",
    "publisher": "",
    "publish_date": "1976-10-01",
    "isbn": "978-4041366042"
  }
}
```

Of course, you can generate the json document by calling `to_json()` many times like this:

```
def output_references():
  user = User.objects(pk=ObjectId("570ee9d1fec55e755db82129")).get()
  user_dct = json.loads(user.to_json())
  user_dct["books"] = [
    json.loads(book.to_json()) for book in user.books_bought
  ]
  user_dct["favorite_one"] = json.loads(user.favorite_one.to_json())
  return jsonify(user_dct)
  # ...And what if there are references in the referenced document??
```

However, as you can see, that code is messy and it has a problem that causes code-bloat. To avoid the problem, this script has a function called `Follow Reference` since version 0.9. To use it, you can just pass `follow_reference=True` to `to_json` function like this:

```
def output_references():
  user = User.objects(pk=ObjectId("570ee9d1fec55e755db82129")).get()
  return jsonify(json.loads(user.to_json(follow_reference=True)))
```

Note that setting `follow_reference=True`, `Document.to_json` checks the reference recursively until the depth reaches 3rd depth. To change the maximum recursion depth, you can set the value you want to `max_depth`:

```
def output_references():
  user = User.objects(pk=ObjectId("570ee9d1fec55e755db82129")).get()
  return jsonify(json.loads(user.to_json(follow_reference=True, max_depth=5)))
```

CHAPTER 3

Addtional Features

3.1 FollowReferenceField

This script also provides a field that supports serialization of the reference with `follow_reference=True`. Unlike `ReferenceField`, this field supports deserialization and automatic-save.

To use this field, you can just simply declare the field as usual. For example, like this:

```
import mongoengine as db
import mongoengine_goodjson as gj


class User(gj.Document):
  """User info."""
  name = db.StringField()
  email = db.EmailField()

class DetailedProfile(gj.Document):
  """Detail profile of the user."""
  # FollowReferenceField without auto-save
  user = gj.FollowReferenceField(User)
  yob = db.DateTimeField()
  # FollowReferenceField with auto-save
  partner = gj.FollowReferenceField(User, autosave=True)
```

3.2 Exclude fields from JSON serialization/deserialization

Sometimes you might want to exclude fields from JSON serialization, but to do so, you might need to decode JSON-serialized string, pop the key, then, serialize the dict object again. Since 0.11, metadata `exclude_to_json`, `exclude_from_json`, and code:*exclude_json* are available and they exclude field on the following specific actions:

- Setting Truthy value to `exclude_to_json`, the corresponding field is omitted from JSON encoding. Note that this excludes fields JSON encoding only.
- Setting Truthy value to `exclude_from_json`, the corresponding field is omitted from JSON decoding. Note that this excludes fields JSON decoding only.
- Setting Truhy value to `exclude_json`, the corresponding field is omitted from JSON encoding and decoding.

3.2.1 Example

To use the exclusion, you can just put exclude metadata like this:

```
import mongoengine_goodjson as gj
import mongoengine as db


class ExclusionModel(gj.Document):
    """Example Model."""
    to_json_exclude = db.StringField(exclude_to_json=True)
    from_json_exclude = db.IntField(exclude_from_json=True)
    json_exclude = db.StringField(exclude_json=True)
    required = db.StringField(required=True)


def get_json_obj(*q, **query):
    model = Exclude.objects(*q, **query).get()
    # Just simply call to_json :)
    return model.to_json()


def get_json_list(*q, **query):
    # You can also get JSON serialized text from QuerySet.
    return Exclude.objects(*q, **query).to_json()


# Decoding is also simple.
def get_obj_from_json(json_text):
  return Exclude.from_json(json_text)


def get_list_from_json(json_text):
  return Exclude.objects.from_json(json_text)
```

3.3 Reference Limit

Since version 1.0.0, the method to limit recursive depth is implemented.

By default, `to_json` serializes the document until the cursor reaches 3rd level. To change the maximum depth level, change `max_depth` kwargs.

As of 1.1.0, callable function can be set to `max_depth`, and `to_json` calls max_depth with the document that the field holds, and current depth level. If the function that is associated with `max_depth` returns truthy values, the serialization will be stop.

Note that when you use callable `max_depth` of `FollowReferenceField`, the border of the document i.e. the document that `max_depth` returned truthy value, will **NOT** be serialized while `to_json()` does. It just be "id" of

the model.

3.3.1 Code Example

Here is the code example of Limit Recursion:

```
import mongoengine as db
import mongoengine_goodjson as gj


class User(gj.Document):
  """User info."""
  name = db.StringField()
  email = db.EmailField()
  # i.e. You can access everyone in the world by Six Degrees of Separation
  friends = db.ListField(gj.FollowReferenceField("self", max_depth=6))

  # If the name of the user is Alice, Mary, or Bob, it will refer more depth.
  not_friend = gj.FollowReferenceField(
    "self", max_depth=lambda doc, cur_depth: doc.name not in [
      "Alice", "Mary", "Bob"
    ]
  )

class DetailedProfile(gj.Document):
  """Detail profile of the user."""
  user = gj.FollowReferenceField(User)
  yob = db.DateTimeField()
```

To disable the limit, put negative number to `max_depth`, however you should make sure that the model has neither circuit nor self-reference.

CHAPTER 4

Dive in the deep

4.1 Encoder / Decoder

Unlike JSON encoder / decoder at pytmongo, `mongoengine_goodjson` passes encoder / decoder to json.dump and json.load by using `cls` and `object_hook`. Therefore, passing `args` or `kwargs` to `mongoengine_goodjson.Document.to_json` / `mongoengine_goodjson.Document.from_json`, The arguments are put into json.dump and json.load.

4.1.1 Code Example

Here's the example code what this section is saying. In this code, the document tries to serialize date into epoch time format (not ISO format).

```
import mongoengine as db
import mongoengine_goodjson as gj


class User(gj.Document):
  """User class."""
  name = db.StringField(required=True, unique=True)
  registered_date = db.DateTimeField()

  def to_json(self, *args, **kwargs):
    """Serialize into json."""
    return super(User, self).to_json(epoch_mode=True)
```

4.2 FAQ from issue tracker

Q: I'm using third-party package such as flask-mongoengine, but no ObjectId is replaced (#34)

A: Some third-party package has abstract classes that inherit classes from MongoEngine. To use `mongoengine_goodjson` with those packages, you will need to inherit the both of documents and queryset.

4.2.1 Example Code

Here is the example code to solve inheritance problem.

```
import mongoengine as db
import flask_mongoengine as fm
import mongoengine_goodjson as gj

class QuerySet(fm.BaseQuerySet, gj.QuerySet):
  """Queryset."""
  pass


class Document(db.Document, gj.Document):
  """Document."""
  meta = {
    'abstract': True,
    'queryset_class': QuerySet
  }


class User(Document):
  """User class."""
  name = db.StringField(required=True, unique=True)
  registered_date = db.DateTimeField()
```

Q: Is there a way to specify which format a DatetimeField will be resolved to? (#38)

A: Check *Encoder / Decoder*

CHAPTER 5

MIT License

CHAPTER 6

Contribution

This document describes how to contribute to this project.

6.1 Posting Issues

Posting issue is appreciated. If you found issues, you can report it to issues tracker. Note that you will need GitHub account before you send the bug report.

When you try to create an issue, you will see the page to choice the type of the issue you want to create and please choose one of the type. You can also open regular issue, but describe your issue / question in detail.

6.2 Solve your problem by your hand

This package is distributed as an Open-Source Software under the terms of MIT License. Hence, you are able to change the code of this package.

6.3 Testbed Environment

As you can see the package, this code is using tox that can test multiple-version of Python. In particular, this package is using the lates version of Python 3 and Python 2. To test this package, install packages in requirements.txt like this:

```
$ pip install -r requirements.txt
```

If you have pip-tools, you can also install the package by doing this:

```
$ pip-sync
```

To keep your system site package clean, using venv is recommended. To use it, you can make virtual environment before installing the packages:

```
$ python -m venv venv && source ./venv/bin/activate
```

Then, run the pip.

To test the code, you can use tox that is installed by pip:

```
(venv)$ tox
# or...
(venv)$ tox -p all
# or...
(venv)$ tox -p auto
```

6.4 Pull Request

If you have coding skills and time to fix your problem, please create a Pull Request. This is much more appreciated than Posting Issues.

6.4.1 Before sending pull request

Sending pull request is very appreciated, but please note:

- **Test code is mandatory.** Your bug must be reproducible, and writing test code is showing the proof of the bug. Any pull requests that don't have test code might be rejected.
- **Not all pull request is merged.** Your pull request is not always accepted and/or merged. However, Hiro absolutely appreciate your contribution.

Note pull requests that don't follow the above rule might not be merged.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search