Eve-Mongoengine Documentation

Release 0.0.7

Stanislav Heller

October 07, 2014

Contents

Eve-Mongoengine is and Eve extension, which enables Mongoengine ODM models to be used as eve schema. If you use mongoengine in your application and simultaneously want to use eve, instead of writing schema again in cerberus format, you can use this extension, which takes your mongoengine models and auto-transforms it into creberus schema.

Contents:

CHAPTER 1

Features

Main features:

- Auto-generated schema out of your mongoengine models
- Every operation goes through mongoengine -> you do not loose your mongoengine hooks
- Support for most of mongoengine fields (see Limitations for more info)
- Mongoengine validation layer not disconnected - use it as you wish

1.1 Validation

By default, eve validates against cerberus schema. Because mongoengine has larger scale of validation possiblities, there are some cases, when cerberus is not enough. Eve-Mongoengine comes with fancy solution: all errors, which are catchable by cerberus, are catched by cerberus and mongoengine ones are catched by custom validator and returned in cerberus error format. Example of this case could be mongoengine's URLField, which does not have it's cerberus opposie. In this case, if you fill in wrong URL, you get mongoengine error message. Let's see an example with internet resource as a model:

```
class Resource(Document):
    url = URLField()
    author = StringField()
```

And then if you make POST request with wrong URL:

```
$ curl -d '{"url": "not-an-url", "author": "John"}' -H 'Content-Type: application/json' http://my-eve
```

The response will contain:

```
{"_status": "ERR", "_issues": {'url': "ValidationError (Resource:None) (Invalid URL: not-an-url: ['u
```

1.2 Advanced model registration

If you want to use the name of model class "as is", use option `lowercase=False` in `add_model()` method:

```
ext.add_model(Person, lowercase=False)
```

Then you will have to ask the server for `/Person/` URL.

In `add_model()` method you can add every possible parameter into resource settings. Even if you want to overwrite some settings, which generates eve-mongoengine under the hood, you can overwrite it this way:

```
ext.add_model(Person,                                          # model or models
              resource_methods=['GET'],                        # allow only GET
              cache_control="max-age=600; must-revalidate") # set max-age
```

When you register more than one model at time, you need to encapsulate all models into list:

```
ext.add_model([Person, Car, House, Dog])
```

HTTP Methods

By default, all HTTP methods are allowed for registered classes:

- resource methods: *GET, POST, DELETE*
- item methods: *GET, PATCH, PUT, DELETE*

1.3 About mongoengine fields

Because Eve contains default functionality, which maintains fields 'updated' and 'created', there has to be special hacky way how to do it in mongoengine too. At the time of initializing EveMongoengine extension, all registered mongoengine classes get two new fields: `updated` and `created`, both type mongoengine.DateTimeField (of course field names are taken from config values `LAST_UPDATED` and `DATE_CREATED`. This is is the only way how to ensure, that Eve will have these fields avaliable for storing it's information about entity. So please, do not be surprised, that there are two more fields in your model class:

```
class Person(mongoengine.Document):
    name = mongoengine.StringField()
    age = mongoengine.IntField()

app = Eve()
ext = EveMongoengine(app)
ext.add_model(Person)

# Note that in db there are attributes '_updated' and '_created'.
# Mongoengine field names are without underscore prefix!
Person._fields.keys() # equals ['name', 'age', 'updated', 'created']
```

If you already have these fields in your model, Eve will probably scream at you, that it's not possible to have these fields in schema.

1.4 User-defined fields

Sometimes you want to use your special kind of field, say `HelloField`, which is build for your specific purpose. So how does eve-mongoengine know how to deal with it when you register a class with that field? Let have a look...

Say `HelloField` is defined like this:

```
from mongoengine import StringField

class HelloField(StringField):
    """
    Allmighty string field, which counts number of 'Hello's in the
    string and throws exception if there are less than 2 'Hello's.
    """
```

Fancy, right? :) Well...not at all. But as an example it's fine. So what eve-mongoengine does with this? It checks the field and recognizes that it's non-standard field class. Then it looks into class bases and tries to get some known field class out of there. In this example the field will be considered as `{'type': 'string'}`. Simple.

Keep in mind that eve-mongoengine knows how to deal with only these fields, which are derived from non-BaseField classes, everything other will be considered as `DynamicField`.

1.5 Mongoengine hooks

If you use mongoengine hooks, you may be interested in what call is performed when POSTing documents or what kind of call is being executed while performing any other method from Eve's REST API. Here is the list you need:

HTTP method	mongoengine's API call
GET resource	`QuerySet.filter()` + `only()`, `exclude()`, `limit()`, `skip()`, `order_by()`
GET item	`QuerySet.get()` (+ every filtering and limiting methods)
POST item	`Document.save()`
PUT item	`Document.save()`
PATCH item	`QuerySet.update_one()` (atomic)
DELETE item	`QuerySet.delete()`

So if you have some hook bound to `save()` method, it should be executed every POST and PUT call you make using Eve. But you have an option to use `save()` method in `PATCH` requests in exchange for one database fetch, so it is relatively slower. If you want to use this feature, set this options in data layer:

```
app = Eve()
ext = EveMongoengine(app)
#: this switches from using QuerySet.update_one() to Document.save()
app.data.mongoengine_options['use_atomic_update_for_patch'] = False
ext.add_model(Person)
```

1.6 Limitations

- You have to give Eve some dummy domain to shut him up. Without this he will complain about empty domain.
- You cannot use mongoengine's custom `primary_key` (because of Eve).
- Cannot use `GenericEmbeddedDocumentField`, `SequenceField`.
- Tested only on python 2.7 and 3.3.
- If you update your document using mongoengine model (i.e. by calling `save()`, the `updated` field wont be updated to current time. This is because there arent any hooks bound to `save()` or `update()` methods and I consider this evil.

CHAPTER 2

API Documentation

2.1 eve_mongoengine

This module implements Eve extension which enables Mongoengine models to be used as eve schema. If you use mongoengine in your application and simultaneously want to use eve, instead of writing schema again in cerberus format, you can use this extension, which takes your mongoengine models and auto-transforms it into creberus schema.

copyright

3. 2014 by Stanislav Heller.

license BSD, see LICENSE for more details.

class `eve_mongoengine.__init__.`**`EveMongoengine`**(*app=None*)

Bases: `object`

An extension to Eve which allows Mongoengine models to be registered as an Eve's "domain".

Acts as Flask extension and implements its 'protocol'.

Usage:

```
from eve_mongoengine import EveMongoengine
from eve import Eve

app = Eve()
ext = EveMongoengine(app)
ext.add_model([MyModel, MySuperModel])
```

This class tries hard to be extendable and hackable as possible, every possible value is either a method param (for IoC-DI) or class attribute, which can be overwriten in subclass.

`add_model`(*models*, *lowercase=True*, ***settings*)

Creates Eve settings for mongoengine model classes.

Returns dict which has to be passed to the settings param in Eve's constructor.

Parameters

- **model** – model or list of them (subclasses of `mongoengine.Document`).
- **lowercase** – if true, all class names will be taken lowercase as resource names. Default True.
- **settings** – any other keyword argument will be treated as param to settings dictionary.

datalayer_class
Datalayer class - instance of this class is pushed to app.data attribute and Eve does it's magic. See `datalayer.MongoengineDataLayer` for more info.

alias of `MongoengineDataLayer`

default_item_methods = ['GET', 'PATCH', 'PUT', 'DELETE']
Default HTTP methods allowed to manipulate with items (single records). These are assigned to settings of every registered model, if not given others.

default_resource_methods = ['GET', 'POST', 'DELETE']
Default HTTP methods allowed to manipulate with whole resources. These are assigned to settings of every registered model, if not given others.

fix_model_class(*model_cls*)
Internal method invoked during registering new model.

Adds necessary fields (updated and created) into model class to ensure Eve's default functionality.

This is a helper for correct manipulation with mongoengine documents within Eve. Eve needs 'updated' and 'created' fields for it's own purpose, but we cannot ensure that they are present in the model class. And even if they are, they may be of other field type or missbehave.

Parameters model_cls – mongoengine's model class (instance of subclass of `mongoengine.Document`) to be fixed up.

init_app(*app*)
Binds EveMongoengine extension to created eve application.

Under the hood it fixes all registered models and overwrites default eve's datalayer `eve.io.mongo.Mongo` into `eve_mongoengine.datalayer.MongoengineDataLayer`.

This method implements flask extension interface: :param app: eve application object, instance of `eve.Eve`.

schema_mapper_class
Mapper from mongoengine model into cerberus schema. This class may be subclassed in the future to support new mongoenigne's fields.

alias of `SchemaMapper`

settings_class
The class used as settings dictionary. Usually subclass of dict with tuned methods/behaviour.

alias of `Settings`

validator_class
The class used as Eve validator, which is also one of Eve's constructor params. In EveMongoengine, we need to overwrite it. If extending, assign only subclasses of `EveMongoengineValidator`.

alias of `EveMongoengineValidator`

`eve_mongoengine.__init__.`**fix_last_updated**(*sender*, *document*, ***kwargs*)
Hook which updates LAST_UPDATED field before every Document.save() call.

`eve_mongoengine.__init__.`**get_utc_time**()
Returns current datetime in system-wide UTC format wichout microsecond part.

2.2 eve_mongoengine.schema

Mapping mongoengine field types to cerberus schema.

copyright

3. 2014 by Stanislav Heller.

license BSD, see LICENSE for more details.

class `eve_mongoengine.schema.`**`SchemaMapper`**
Bases: `object`

Default mapper from mongoengine model classes into cerberus dict-like schema.

classmethod `create_schema` (*model_cls*, *lowercase=True*)

Parameters

- **model_cls** – Mongoengine model class, subclass of `mongoengine.Document`.
- **lowercase** – True if names of resource for model class has to be treated as lowercase string of classname.

classmethod `get_subresource_settings` (*model_cls*, *resource_name*, *resource_settings*, *lowercase=True*)
Yields name of subresource domain and it's settings.

classmethod `process_field` (*field*, *lowercase*)
Returns Eve field definition from Mongoengine field

Parameters

- **field** – Mongoengine field
- **lowercase** – True if names of resource for model class has to be treated as lowercase string of classname.

2.3 eve_mongoengine.validation

This module implements custom validator based on `eve.io.mongo.validation`, which is cerberus-validator extension.

The purpose of this module is to enable validation for special mongoengine fields.

copyright

3. 2014 by Stanislav Heller.

license BSD, see LICENSE for more details.

class `eve_mongoengine.validation.`**`EveMongoengineValidator`**(*schema*, *resource=None*)
Bases: `eve.io.mongo.validation.Validator`

Helper validator which adapts mongoengine special-purpose fields to cerberus validator API.

`validate`(*document*, *schema=None*, *update=False*, *context=None*)
Main validation method which simply tries to validate against cerberus schema and if it does not fail, repeats the same against mongoengine validation machinery.

2.4 eve_mongoengine.struct

Datastructures for eve-mongoengine.

copyright

3. 2014 by Stanislav Heller.

license BSD, see LICENSE for more details.

2.5 eve_mongoengine.datalayer

This module implements eve's data layer which uses mongoengine models instead of direct pymongo access.

copyright

3. 2014 by Stanislav Heller.

license BSD, see LICENSE for more details.

class `eve_mongoengine.datalayer.`**`MongoengineDataLayer`**(*ext*)
Bases: `eve.io.mongo.mongo.Mongo`

Data layer for eve-mongoengine extension.

Most of functionality is copied from `eve.io.mongo.Mongo`.

`default_queryset` = **'objects'**
name of default queryset, where datalayer asks for data

`find`(*resource*, *req*, *sub_resource_lookup*)
Seach for results and return list of them.

Parameters

- **resource** – name of requested resource as string.
- **req** – instance of `eve.utils.ParsedRequest`.
- **sub_resource_lookup** – sub-resource lookup from the endpoint url.

`find_one`(*resource*, *req*, ***lookup*)
Look for one object.

`insert`(*resource*, *doc_or_docs*)
Called when performing POST request

`json_encoder_class`
default JSON encoder

alias of `MongoengineJsonEncoder`

`mongoengine_options` = **{'use_atomic_update_for_patch': True}**
Options for usage of mongoengine layer. use_atomic_update_for_patch - when set to True, Mongoengine layer will use update_one() method (which is atomic) for updating. But then you will loose your pre/post-save hooks. When you set this to False, for updating will be used save() method.

`remove`(*resource*, *lookup*)
Called when performing DELETE request.

`replace`(*resource*, *id_*, *document*)
Called when performing PUT request.

`update`(*resource*, *id_*, *updates*)
Called when performing PATCH request.

class eve_mongoengine.datalayer.**MongoengineJsonEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *item_sort_key=None*, *for_json=False*, *ignore_nan=False*, *int_as_string_bitcount=None*)

Bases: eve.io.mongo.mongo.MongoJSONEncoder

Propretary JSON encoder to support special mongoengine's special fields.

default(*obj*)

class eve_mongoengine.datalayer.**PymongoQuerySet**(*qs*)

Bases: object

Dummy mongoenigne-like QuerySet behaving just like queryset with as_pymongo() called, but returning ALL fields in subdocuments (which as_pymongo() somehow filters).

CHAPTER 3

Install

Simple installation using pip:

```
pip install eve-mongoengine
```

It loads all dependencies as well (Eve and nothing more!).

For development use virtualenv and editable copy of repisotory:

```
pip install -e git+https://github.com/hellerstanislav/eve-mongoengine#egg=eve-mongoengine
```

CHAPTER 4

Usage

```
import mongoengine
from eve import Eve
from eve_mongoengine import EveMongoengine

# create some dummy model class
class Person(mongoengine.Document):
    name = mongoengine.StringField()
    age = mongoengine.IntField()

# default eve settings
my_settings = {
    'MONGO_HOST': 'localhost',
    'MONGO_PORT': 27017,
    'MONGO_DBNAME': 'eve_mongoengine_test'
}

# init application
app = Eve(settings=my_settings)
# init extension
ext = EveMongoengine(app)
# register model to eve
ext.add_model(Person)

# let's roll
app.run()
```

Or, if you are setting up your data before Eve is initialized, as is the case with application factories:

:: import mongoengine from eve import Eve from eve_mongoengine import EveMongoengine

ext = EveMongoengine() ... # init application app = Eve(settings=my_settings)

init extension ext.init_app(app) ...

e

U

V