Anyblok / Marshmallow Documentation

Release 1.4.0

Jean-Sébastien SUZANNE

Apr 07, 2018

Contents

Contents

CHAPTER 1

Front Matter

Information about the AnyBlok / Marshmallow project.

1.1 Project Homepage

AnyBlok is hosted on github - the main project page is at https://github.com/AnyBlok/AnyBlok_Marshmallow. Source code is tracked here using GIT.

Releases and project status are available on Pypi at http://pypi.python.org/pypi/anyblok_marshmallow.

The most recent published version of this documentation should be at http://doc.anyblok-marshmallow.anyblok.org.

1.2 Project Status

AnyBlok with Marshmallow is currently in beta status and is expected to be fairly stable. Users should take care to report bugs and missing features on an as-needed basis. It should be expected that the development version may be required for proper implementation of recently repaired issues in between releases;

1.3 Installation

Install released versions of AnyBlok from the Python package index with pip or a similar tool:

```
pip install anyblok_marshmallow
```

Installation via source distribution is via the `setup.py` script:

```
python setup.py install
```

Installation will add the `anyblok` commands to the environment.

1.4 Unit Test

Run the test with `nose`:

```
pip install nose
nosetests anyblok_marshmallow/tests
```

1.5 Dependencies

AnyBlok works with **Python 3.3** and later. The install process will ensure that AnyBlok, marshmallow and marshmallow-sqlalchemy are installed, in addition to other dependencies. The latest version of them is strongly recommended.

1.6 Contributing (hackers needed!)

Anyblok / Marshmallow is at a very early stage, feel free to fork, talk with core dev, and spread the word!

1.7 Author

Jean-Sébastien Suzanne

1.8 Contributors

Anybox team:

- Jean-Sébastien Suzanne

Sensee team:

- Franck Bret

1.9 Bugs

Bugs and feature enhancements to AnyBlok should be reported on the Issue tracker.

Contents

        * *Add the **registry** by the Meta*
        * *Add the **registry** during init*
        * *Add the **registry** by the context*
        * *Add the **registry** when the de(serialization or validatoris called*
    - ***model** option*
    - ***only_primary_key** option*

CHAPTER 2

Memento

2.1 Declare your AnyBlok model

```
from anyblok.column import Integer, String
from anyblok.relationship import Many2One, Many2Many
from anyblok import Declarations


@Declarations.register(Declarations.Model)
class City:

    id = Integer(primary_key=True)
    name = String(nullable=False)
    zipcode = String(nullable=False)

    def __repr__(self):
        return '<City(name={self.name!r})>'.format(self=self)


@Declarations.register(Declarations.Model)
class Tag:

    id = Integer(primary_key=True)
    name = String(nullable=False)

    def __repr__(self):
        return '<Tag(name={self.name!r})>'.format(self=self)


@Declarations.register(Declarations.Model)
class Customer:
    id = Integer(primary_key=True)
    name = String(nullable=False)
    tags = Many2Many(model=Declarations.Model.Tag)
```

(continues on next page)

(continued from previous page)

```
    def __repr__(self):
        return '<Customer(name={self.name!r}, '
               'tags={self.tags!r})>'.format(self=self)


@Declarations.register(Declarations.Model)
class Address:

    id = Integer(primary_key=True)
    street = String(nullable=False)
    city = Many2One(model=Declarations.Model.City, nullable=False)
    customer = Many2One(
        model=Declarations.Model.Customer, nullable=False,
        one2many="addresses")
```

Warning: The **AnyBlok model** must be declared in a blok

2.2 Declare your schema

```
from anyblok_marshmallow import ModelSchema, PostLoadSchema, Nested

class CitySchema(ModelSchema):

    class Meta:
        model = 'Model.City'


class TagSchema(ModelSchema):

    class Meta:
        model = 'Model.Tag'


class AddressSchema(ModelSchema):

    # follow the relationship Many2One and One2One
    city = Nested(CitySchema)

    class Meta:
        model = 'Model.Address'


class CustomerSchema(PostLoadSchema, ModelSchema):

    # follow the relationship One2Many and Many2Many
    # - the many=True is required because it is *2Many
    # - exclude is used to forbid the recurse loop
    addresses = Nested(AddressSchema, many=True, exclude=('customer', ))
    tags = Nested(TagSchema, many=True)

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        model = 'Model.Customer'
        # optionally attach an AnyBlok registry
        # to use for serialization, desarialization and validation
        registry = registry


customer_schema = CustomerSchema()
```

Note: **New** in version **1.1.0** the Nested field must come from **anyblok_marshmallow**, because **marshmallow** cache the Nested field with the context. And the context is not propagated again if it changed

Note: **Ref** in version **1.4.0**, `post_load_return_instance` was replaced by the mixin class `PostLoadSchema`

2.3 (De)serialize your data and validate it

```
customer = registry.Customer.insert(name="JS Suzanne")
tag1 = registry.Tag.insert(name="tag 1")
customer.tags.append(tag1)
tag2 = registry.Tag.insert(name="tag 2")
customer.tags.append(tag2)
rouen = registry.City.insert(name="Rouen", zipcode="76000")
paris = registry.City.insert(name="Paris", zipcode="75000")
registry.Address.insert(customer=customer, street="Somewhere", city=rouen)
registry.Address.insert(customer=customer, street="Another place", city=paris)

dump_data = customer_schema.dump(customer).data
# {
#     'id': 1,
#     'name': 'JS Suzanne',
#     'tags': [
#         {
#             'id': 1,
#             'name': 'tag 1',
#         },
#         {
#             'id': 2,
#             'name': 'tag 2',
#         },
#     ],
#     'addresses': [
#         {
#             'id': 1
#             'street': 'Somewhere'
#             'city': {
#                 'id': 1,
#                 'name': 'Rouen',
#                 'zipcode': '76000',
#             },
#         },
```

(continues on next page)

(continued from previous page)

```
#         {
#             'id': 2
#             'street': 'Another place'
#             'city': {
#                 'id': 2,
#                 'name': 'Paris',
#                 'zipcode': '75000',
#             },
#         },
#     ],
# }

customer_schema.load(dump_data).data
# <Customer(name='JS Suzanne' tags=[<Tag(name='tag 1')>, <Tag (name='tag 2')>])>

errors = customer_schema.validate(dump_data)
# dict with all the validating errors
```

Note: We have an instance of the model cause of the mixin `PostLoadSchema`

2.4 Give the registry

The schema need to have the registry.

If no registry found when the de(serialization) or validation then the **RegistryNotFound** exception will be raised.

2.4.1 Add the registry by the Meta

This is the solution given in the main exemple:

```
class CustomerSchema(ModelSchema):

    class Meta:
        model = 'Model.Customer'
        registry = registry
```

2.4.2 Add the registry during init

This solution is use during the instanciation

```
customer_schema = CustomerSchema(registry=registry)
```

2.4.3 Add the registry by the context

This solution is use during the instanciation or after

```
customer_schema = CustomerSchema(context={'registry': registry})
```

or

```
customer_schema = CustomerSchema()
customer_schema.context['registry'] = registry
```

2.4.4 Add the registry when the de(serialization or validatoris called

```
customer_schema.dump(customer, registry=registry)
customer_schema.load(dump_data, registry=registry)
customer_schema.validate(dump_data, registry=registry)
```

2.5 model option

This option add in the model name. As the registry, this option can be passed by definition, initialization, context or during the call of the (de)serialization / validation

```
class AnySchema(ModelSchema):

    class Meta:
        model = "Model.Customer"
```

or

```
any_schema = AnySchema(model="Model.customer")
```

or

```
any_schema.context['model'] = "Model.Customer"
```

or

```
any_schema.dump(instance, model="Model.Customer")
any_schema.load(dump_data, model="Model.Customer")
any_schema.validate(dump_data, model="Model.Customer")
```

2.6 only_primary_key option

This option add in the only argument the primary keys of the model. As the registry, this option can be passed by definition, initialization, context or during the call of the (de)serialization / validation

```
class CustomerSchema(ModelSchema):

    class Meta:
        model = "Model.Customer"
        only_primary_key = True
```

or

```
customer_schema = CustomerSchema(only_primary_key=True)
```

or

```
customer_schema.context['only_primary_key'] = True
```

or

```
customer_schema.dump(instance, only_primary_key=True)
customer_schema.load(dump_data, only_primary_key=True)
customer_schema.validate(dump_data, only_primary_key=True)
```

Contents

CHAPTER 3

Exceptions

3.1 RegistryNotFound

exception anyblok_marshmallow.exceptions.**RegistryNotFound**

Bases: Exception

Exception raised when no registry is found to build schema

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER 4

Fields

4.1 Nested

class anyblok_marshmallow.fields.**Nested**(*nested*, *default=<marshmallow.missing>*, *exclude=()*, *only=None*, ***kwargs*)

Bases: marshmallow.fields.Nested

Inherit marshmallow fields.Nested

context
The context dictionary for the parent Schema.

deserialize(*value*, *attr=None*, *data=None*)
Deserialize value.

Raises **ValidationError** – If an invalid value is passed or if a required value is missing.

fail(*key*, ***kwargs*)
A helper method that simply raises a *ValidationError*.

get_value(*attr*, *obj*, *accessor=None*, *default=<marshmallow.missing>*)
Return the value for a given key from an object.

root
Reference to the *Schema* that this field belongs to even if it is buried in a *List*. Return *None* for unbound fields.

schema
Overload the super property to remove cache

it is the only way to propagate the context at each call

serialize(*attr*, *obj*, *accessor=None*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** (*str*) – The attibute or key to get from the object.

- **obj** (*str*) – The object to pull the key from.
- **accessor** (*callable*) – Function used to pull values from `obj`.

Raises **ValidationError** – In case of formatting problem

CHAPTER 5

Schema

5.1 update_from_kwargs

anyblok_marshmallow.schema.**update_from_kwargs**(**entries*)
decorator to get temporaly the value in kwargs and put it in schema

Params entries array ok entry name to take from the kwargs

5.2 format_field

anyblok_marshmallow.schema.**format_fields**(*x*)
remove the anyblok prefix form the field name

5.3 ModelConverter

class anyblok_marshmallow.schema.**ModelConverter**(*schema_cls=None*)
Bases: marshmallow_sqlalchemy.convert.ModelConverter

Overwrite the ModelConverter class of marshmallow-sqlalchemy

The goal if to fix the fieldname, because they are prefixed.

fields_for_model(*Model*, ***kwargs*)
Overwrite the method and remove prefix of the field name

5.4 ModelSchemaOpts

class anyblok_marshmallow.schema.**ModelSchemaOpts**(*meta*, **args*, ***kwargs*)
Bases: marshmallow.schema.SchemaOpts

Model schema option for Model schema

Add get option from the Meta:

- model: name of an AnyBlok model **required**
- registry: an AnyBlok registry

5.5 ModelSchema

class anyblok_marshmallow.schema.**ModelSchema**(**args*, ***kwargs*)
Bases: marshmallow.schema.Schema

A marshmallow schema based on the AnyBlok Model

Wrap the real schema, because at the instanciation the registry is not available

class Meta
Bases: object

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- fields: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the** explicitly declared fields. additional and fields are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **exclude: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- **dateformat: Date format for all DateTime fields that do not have their** date format explicitly specified.
- **strict: If *True*, raise errors during marshalling rather than** storing them.
- **json_module: JSON module to use for *loads* and *dumps*.** Defaults to the json module in the stdlib.
- **ordered: If *True*, order serialization output according to the** order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **index_errors: If *True*, errors dictionaries will include the index** of invalid items in a collection.
- load_only: Tuple or list of fields to exclude from serialized results.
- dump_only: Tuple or list of fields to exclude from deserialization

OPTIONS_CLASS
alias of ModelSchemaOpts

classmethod accessor(*func*)
Decorator that registers a function for pulling values from an object to serialize. The function receives the `Schema` instance, the `key` of the value to get, the `obj` to serialize, and an optional `default` value.

Deprecated since version 2.0.0: Set the `error_handler` class Meta option instead.

dumps(*obj*, *many=None*, *update_fields=True*, **args*, ***kwargs*)
Same as `dump()`, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** (*bool*) – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **update_fields** (*bool*) – Whether to update the schema's field classes. Typically set to *True*, but may be *False* when serializing a homogenous collection. This parameter is used by *fields.Nested* to avoid multiple updates.

Returns A tuple of the form (`data`, `errors`)

Return type *MarshalResult*, a *collections.namedtuple*

New in version 1.0.0.

classmethod error_handler(*func*)
Decorator that registers an error handler function for the schema. The function receives the `Schema` instance, a dictionary of errors, and the serialized object (if serializing data) or data dictionary (if deserializing data) as arguments.

Example:

```
class UserSchema(Schema):
    email = fields.Email()

@UserSchema.error_handler
def handle_errors(schema, errors, obj):
    raise ValueError('An error occurred while marshalling {}'.format(obj))

user = User(email='invalid')
UserSchema().dump(user)  # => raises ValueError
UserSchema().load({'email': 'bademail'})  # raises ValueError
```

New in version 0.7.0.

Deprecated since version 2.0.0: Set the `error_handler` class Meta option instead.

generate_marsmallow_instance()
Generate the real mashmallow-sqlalchemy schema

get_attribute(*attr*, *obj*, *default*)
Defines how to pull values from an object to serialize.

New in version 2.0.0.

handle_error(*error*, *data*)
Custom error handler function for the schema.

Parameters

- **error** (*ValidationError*) – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.

New in version 2.0.0.

loads(*json_data*, *many=None*, **args*, ***kwargs*)
Same as `load()`, except it takes a JSON string as input.

Parameters

- **json_data** (*str*) – A JSON string of the data to deserialize.
- **many** (*bool*) – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** (*bool|tuple*) – Whether to ignore missing fields. If *None*, the value for *self.partial* is used. If its value is an iterable, only missing fields listed in that iterable will be ignored.

Returns A tuple of the form (`data`, `errors`)

Return type *UnmarshalResult*, a *collections.namedtuple*

New in version 1.0.0.

on_bind_field(*field_name*, *field_obj*)
Hook to modify a field when it is bound to the *Schema*. No-op by default.

schema
property to get the real schema

Contents

CHAPTER 6

CHANGELOG

6.1 1.4.0 (2018-04-07)

- Replace **post_load_return_instance** method by **PostLoadSchema** class
- In the case of the field **Selection**, the validator **OneOf** is applied with the available values come from the AnyBlok columns
- Replace **marshmallow_sqlalchemy.fields.Related** by **anyblok_marshmallow.fields.Nested**. The goal is to improve the consistent between all field in the schema

6.2 1.3.0 (2017-12-23)

- [ADD] unittest on some case
- [FIX] AnyBlok field.Function is return as MarshMallow fields.Raw
- [ADD] fields.File, type to encode and decode to/from base 64

6.3 1.2.0 (2017-11-30)

- [REF] decrease complexity
- [IMP] Add `validates_schema` on ModelSchema to automaticly check if the field exist on the model

6.4 1.1.0 (2017-11-02)

- Add option put only the primary keys
- Fix the Front page

- REF model option, can be given by another way than Meta
- Put RegistryNotFound in exceptions
- Add Nested field, this field is not and have not to be cached

6.5 1.0.2 (2017-10-25)

- Fix pypi documentation

6.6 1.0.0 (2017-10-24)

- Add marshmallow schema for AnyBlok for:
  - Serialization
  - Deserialization
  - Validation

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

A