Flask-MongoEngine Documentation

Release 1.0.1.dev289+gd452613

Ross Lawley

Aug 16, 2022

CONTENTS

A Flask extension that provides integration with MongoEngine. For more information on MongoEngine please check out the MongoEngine Documentation.

It handles connection management for your app. You can also use WTForms as model forms for your models.

CHAPTER

ONE

FLASK CONFIGURATION

Flask-mongoengine does not provide any configuration defaults. User is responsible for setting up correct database settings, to exclude any possible misconfiguration and data corruption.

There are several options to set connection. Please note, that all except recommended are deprecated and may be removed in future versions, to lower code base complexity and bugs. If you use any deprecated connection settings approach, you should update your application configuration.

By default, flask-mongoengine open the connection when extension is instantiated, but you can configure it to open connection only on first database access by setting the `'connect'` dictionary parameter or its `MONGODB_CONNECT` flat equivalent to `False`.

Note: Due lack of developers we are unable to answer/solve not recommended connection methods errors. Please switch to recommended method before posting any issue. Thank you.

1.1 Recommended: List of dictionaries settings

Recommended way for setting up connections is to set `MONGODB_SETTINGS` in you application config. `MONGODB_SETTINGS` is a list of dictionaries, where each dictionary is configuration for individual database (for systems with multi-database) use.

Each dictionary in `MONGODB_SETTINGS` will be passed to `mongoengine.connect()`, which will bypass settings to `mongoengine.register_connection()`. All settings related to `mongoengine.connect()` and `mongoengine.register_connection()` will be extracted by mentioned functions, any other keyword arguments will be silently followed to `pymongo.mongo_client.MongoClient`.

This allows complete and flexible database configuration.

Example:

```
import flask
from flask_mongoengine import MongoEngine

db = MongoEngine()
app = flask.Flask("example_app")
app.config["MONGODB_SETTINGS"] = [
    {
        "db": "project1",
        "host": "localhost",
        "port": 27017,
        "alias": "default",
```

(continues on next page)

(continued from previous page)

```
    }
]
db.init_app(app)
```

1.2 Deprecated: Passing database configuration to MongoEngine class

Deprecated since version 2.0.0.

You can pass database dictionary of dictionaries directly to *MongoEngine* class initialization. Lists of settings not supported.

```
import flask
from flask_mongoengine import MongoEngine

db_config = {
    "db": "project1",
    "host": "localhost",
    "port": 27017,
    "alias": "default",
}
db = MongoEngine(config=db_config)
app = flask.Flask("example_app")

db.init_app(app)
```

1.3 Deprecated: Passing database configuration to MongoEngine.init_app method

Deprecated since version 2.0.0.

You can pass database dictionary of dictionaries directly to *flask_mongoengine.MongoEngine.init_app()* class initialization. Lists of settings not supported.

```
import flask
from flask_mongoengine import MongoEngine

db_config = {
    "db": "project1",
    "host": "localhost",
    "port": 27017,
    "alias": "default",
}
db = MongoEngine()
app = flask.Flask("example_app")

db.init_app(app, config=db_config)
```

1.4 Deprecated: MONGODB_ inside MONGODB_SETTINGS dictionary

Deprecated since version 2.0.0.

Flask-mongoengine will cut off `MONGODB_` prefix from any parameters, specified inside `MONGODB_SETTINGS` dictionary. This is historical behaviour, but may be removed in the future. Providing such settings may raise config errors, when parent packets implement case-sensitive keyword arguments checks. Check issue #451 for example historical problem.

Currently, we are handling all possible case-sensitive keyword settings and related users errors (based on pymongo 4.1.1), but amount of such settings may be increased in future pymongo versions.

Usage of recommended settings approach completely remove this possible problem.

1.5 Deprecated: URI style settings

Deprecated since version 2.0.0.

URI style connections supported as supply the uri as the `host` in the `MONGODB_SETTINGS` dictionary in `app.config`.

Warning: It is not recommended to use URI style settings, as URI style settings parsed and manupulated in all parent functions/methods. This may lead to unexpected behaviour when parent packages versions changed.

Warning: Database name from uri has priority over name. (MongoEngine behaviour).

If uri presents and doesn't contain database name db setting entirely ignore and db name set to `test`:

```
import flask
from flask_mongoengine import MongoEngine

db = MongoEngine()
app = flask.Flask("example_app")
app.config['MONGODB_SETTINGS'] = {
    'db': 'project1',
    'host': 'mongodb://localhost/database_name'
}
db.init_app(app)
```

1.6 Deprecated: Flat MONGODB_ style configuration settings

Deprecated since version 2.0.0.

Connection settings may also be provided individually by prefixing the setting with `MONGODB_` in the `app.config`:

```
import flask
from flask_mongoengine import MongoEngine
```

(continues on next page)

(continued from previous page)

```
db = MongoEngine()
app = flask.Flask("example_app")
app.config['MONGODB_DB'] = 'project1'
app.config['MONGODB_HOST'] = '192.168.1.35'
app.config['MONGODB_PORT'] = 12345
app.config['MONGODB_USERNAME'] = 'webapp'
app.config['MONGODB_PASSWORD'] = 'pwd123'
db.init_app(app)
```

This method does not support multi-database installations.

By default, flask-mongoengine open the connection when extension is instantiated, but you can configure it to open connection only on first database access by setting the `MONGODB_SETTINGS['connect']` parameter or its `MONGODB_CONNECT` flat equivalent to `False`.

CHAPTER

TWO

DATABASE MODEL AND FIELDS DEFINITION

Important: Flask-Mongoengine does not adjust database level behaviour of mongoengine fields definition, except *keyword only definition* requirement. Everything other on database level match mongoengine project. All parent methods, arguments (as keyword arguments) and keyword arguments are supported.

If you are not intend to use WTForms integration, you are free to use fields classes from parent mongoengine project; this should not break anything.

2.1 Supported fields

Flask-Mongoengine support all **database** fields definition. Even if there will be some new field type created in parent mongoengine project, we will silently bypass field definition to it, if we do not declare rules on our side.

Note: Version **2.0.0** Flask-Mongoengine update support mongoengine fields, based on version **mongoengine==0.21**. Any new fields bypassed without modification.

2.2 Keyword only definition

Changed in version 2.0.0.

Database model definition rules and Flask-WTF/WTForms *integration* was seriously updated in version **2.0.0**. Unfortunately, these changes implemented without any deprecation stages.

Before version **2.0.0** Flask-Mongoengine integration allowed to pass fields parameters as arguments. To exclude any side effects or keyword parameters duplication/conflicts, since version **2.0.0** all fields require keyword only setup.

Such approach removes number of issues and questions, when users frequently used Flask-WTF/WTForms definition rules by mistake, or just missed that some arguments was passed to keyword places silently creating unexpected side effects. You can check issue #379 as example of one of such cases.

CHAPTER

THREE

WTFORMS INTEGRATION

Important: Documentation below is related to project version 2.0.0 or higher, old versions has completely different approach for forms generation.

And despite the fact that the old code is included in version 2.0.0 to keep correct deprecation workflow (where possible), it is not documented (and was not) and not maintained.

If you faced any forms problems, consider migration to new methods and approach.

Flask-Mongoengine and Flask-WTF/WTForms are heavily integrated, to reduce amount of boilerplate code, required to make database model and online form. In the same time a lot of options was created to keep extreme flexibility.

After database model definition user does not require to repeat same code in form definition, instead it is possible to use integrated converter, that will do most of the work.

Flask-Mongoengine will transform some model's properties to Flask-WTF/WTForms validators, so user does not need to care about standards. For full list of transformations, please review *global transforms* and specific field documentation below.

In the same time, user is able to adjust database fields definition with specific settings as on stage of Document model definition, as on form generation stage. This allows to create several forms for same model, for different circumstances.

3.1 Requirements

For correct integration behavior several requirements should be met:

- Document classes should be used from Flask-Mongoengine `flask_mongoengine.MongoEngine` class, or from `flask_mongoengine.documents` module.
- Document classes should be used from Flask-Mongoengine `flask_mongoengine.MongoEngine` class, or from `flask_mongoengine.db_fields` module.

3.2 Global transforms

For all fields, processed by Flask-Mongoengine integration:

- If model field definition have `wtf_validators` defined, they will be forwarded to WTForm as `validators`. This is not protection from `validators` extension by Flask-Mongoengine.
- If model field definition have `wtf_filters` defined, they will be forwarded to WTForm as `filters`.
- If model field definition have `required`, then `InputRequired` will be added to form `validators`, otherwise `Optional` added.
- If model field definition have `verbose_name` it will be used as form field `label`, otherwise pure field name used.
- If model field definition have `help_text` it will be used as form field `description`, otherwise empty string used.
- Field's `default` used as form `default`, that's why special WTForms fields implementations was created. Details can be found in *flask_mongoengine.wtf.fields* module. In new form generator only 'Mongo' prefixed classes are used for fields, other classes are deprecated and will be removed in version **3.0.0**. If you have own nesting classes, you should check inheritance and make an update.
- Field's `choices`, if exist, used as form `choices`.

> **Warning:** As at version **2.0.0** there is no `wtf_validators` duplicates/conflicts check. User should be careful with manual `wtf_validators` setup. And in case of forms problems this is first place to look on.
>
> `wtf_validators` and `wtf_filters` duplication check expected in future versions; PRs are welcome.

Some additional transformations are made by specific field, check exact field documentation below for more info.

3.3 BinaryField

Not yet documented. Please help us with new pull request.

3.4 BooleanField

- API: *db_fields.BooleanField*
- Default form field class: *MongoBooleanField*

3.4.1 Form generation behaviour

BooleanField is very complicated in terms of Mongo database support. In Flask-Mongoengine before version **2.0.0+** database BooleanField used `wtforms.fields.BooleanField` as form representation, this raised several not clear problems, that was related to how `wtforms.fields.BooleanField` parse and work with form values. Known problems in version, before **2.0.0+**:

- Default value of field, specified in database definition was ignored, if default is `None` and nulls allowed, i.e. `null=True` (Value was always `False`).

- Field was always created in database document, even if not checked, as there is impossible to split `None` and `False` values, when only checkbox available.

To fix all these issues, and do not create database field by default, Flask-Mongoengine **2.0.0+** uses dropdown field by default.

By default, database BooleanField not allowing `None` value, meaning that field can be `True`, `False` or not created in database at all. If database field configuration allowing `None` values, i.e. `null=True`, then, when nothing selected in dropdown, the field will be created with `None` value.

Important: It is responsobility of developer, to correctly setup database field definition and make proper tests before own application release. BooleanField can create unexpected application behavior in if checks. Developer, should recheck all if checks like:

- `if filed_value:` this will match `True` database value
- `if not filed_value:` this will match `False` or `None` database value or not existing document key
- `if field_value is None:` this will match `None` database value or not existing document key
- `if field_value is True:` this will match `True` database value
- `if field_value is False:` this will match `False` database value
- `if field_value is not None:` this will match `True`, `False` database value
- `if field_value is not True:` this will match `False`, `None` database value or not existing document key
- `if filed_value is not False:` this will match `True`, `None` database value or not existing document key

---

3.4.2 Examples

BooleanField with default dropdown

Such definition will not create any field in document, if dropdown not selected.

```
"""boolean_demo.py"""
from example_app.models import db


class BooleanDemoModel(db.Document):
    """Documentation example model."""

    boolean_field = db.BooleanField()
```

BooleanField with allowed `None` value

Such definition will create document field, even if nothing selected. The value will be `None`. If, during edit, `yes` or `no` dropdown values replaced to `---`, then saved value in document will be also changed to `None`.

By default, `None` value represented as `---` text in dropdown.

```
"""boolean_demo.py"""
from example_app.models import db
```

(continues on next page)

(continued from previous page)

```
class BooleanDemoModel(db.Document):
    """Documentation example model."""

    boolean_field_with_null = db.BooleanField(null=True)
```

BooleanField with replaced dropdown text

Dropdown text can be easily replaced, there is only one requirement: New choices, should be correctly coerced by *coerce_boolean()*, or function should be replaced too.

```
"""boolean_demo.py"""
from example_app.models import db


class BooleanDemoModel(db.Document):
    """Documentation example model."""

    boolean_field_with_as_choices_replace = db.BooleanField(
        wtf_options={
            "choices": [("", "Not selected"), ("yes", "Positive"), ("no", "Negative")]
        }
    )
```

BooleanField with default `True` value, but with allowed nulls

```
"""boolean_demo.py"""
from example_app.models import db


class BooleanDemoModel(db.Document):
    """Documentation example model."""

    true_boolean_field_with_allowed_null = db.BooleanField(default=True, null=True)
```

3.5 ComplexDateTimeField

- API: *db_fields.ComplexDateTimeField*
- Default form field class: `wtforms.fields.DateTimeLocalField`

3.5.1 Form generation behaviour

ComplexDateTimeField stores date and time information in database `string` format. This format allow precision up to microseconds dimension.

Unfortunately, there is no HTML5 field, that allow so high precision. That's why, by default the generated field will use HTML5 <input type="datetime-local"> with precision set to milliseconds.

If you require concrete microseconds for edit purposes, please use `wtforms.fields.DateTimeField` with correct format (see examples below).

Field is easy adjustable, to use any other precision. Check examples and example app for more details.

3.5.2 Examples

dates_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

ComplexDateTimeField with milliseconds precision

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    complex_datetime = db.ComplexDateTimeField()
```

ComplexDateTimeField with seconds precision

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    complex_datetime_sec = db.ComplexDateTimeField(
        wtf_options={"render_kw": {"step": "1"}}
    )
```

ComplexDateTimeField with microseconds precision (text)

```
"""dates_demo.py"""
from wtforms.fields import DateTimeField

from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    complex_datetime_microseconds = db.ComplexDateTimeField(
        wtf_field_class=DateTimeField, wtf_options={"format": "%Y-%m-%d %H:%M:%S.%f"}
    )
```

3.6 DateField

- API: *db_fields.DateField*
- Default form field class: wtforms.fields.DateField

3.6.1 Form generation behaviour

DateField is one of the simplest fields in the forms generation process. By default, the field use wtforms.fields.DateField WTForms class, representing a form input with standard HTML5 <input type="date">. No custom additional transformation done, during field generation. Field is fully controllable by *global transforms*.

3.6.2 Examples

dates_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not limited DateField

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    date = db.DateField()
```

3.7 DateTimeField

- API: *db_fields.DateTimeField*
- Default form field class: `wtforms.fields.DateTimeLocalField`

3.7.1 Form generation behaviour

DateTimeField stores date and time information in database `date` format. This format allow precision up to milliseconds dimension. By default, generated form will use HTML5 <input type="datetime-local"> with precision set to seconds.

Field is easy adjustable, to use any other precision. Check examples and example app for more details.

It is possible to use `wtforms.fields.DateTimeField` for text input behaviour.

3.7.2 Examples

dates_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

DateTimeField with seconds precision

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    datetime = db.DateTimeField()
```

DateTimeField without seconds

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    datetime_no_sec = db.DateTimeField(wtf_options={"render_kw": {"step": "60"}})
```

DateTimeField with milliseconds precision

```
"""dates_demo.py"""
from example_app.models import db


class DateTimeModel(db.Document):
    """Documentation example model."""

    datetime_ms = db.DateTimeField(wtf_options={"render_kw": {"step": "0.001"}})
```

3.8 DecimalField

- API: *db_fields.DecimalField*
- Default form field class: wtforms.fields.DecimalField

3.8.1 Form generation behaviour

From form generation side this field is pretty standard and do not use any form generation adjustments.

If database field definition has any of `min_value` or `max_value`, then NumberRange validator will be added to form field.

3.8.2 Examples

numbers_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not limited DecimalField

```
"""numbers_demo.py"""
from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    decimal_field_unlimited = db.DecimalField()
```

Limited DecimalField

```
"""numbers_demo.py"""
from decimal import Decimal

from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    decimal_field_limited = db.DecimalField(
        min_value=Decimal("1"), max_value=Decimal("200.455")
    )
```

3.9 DictField

- API: *db_fields.DictField*
- Default form field class: *MongoDictField*

DictField has `Object` type in terms of Mongo database itself, so basically it defines document inside document, but without pre-defined structure. That's why this is one of fields, that has default value specified inside Mongoengine itself, and that's why is always (almost) created.

The developer should understand that database keyword argument `default` is forwarded to form by default, but can be separately overwritten in form. This brings a lot of options for form field configuration.

Also, should be additionally noted that database `Null` value in form is represented as empty string. Non-existing field is represented with form `default` for new forms (without instance inside) or with empty string for non-empty forms.

Complicated? Probably. That's why this field was completely rewritten in version **2.0.0**. Check examples, and everything will be clear.

3.9.1 Form generation behaviour

Our default form generation follow Mongoengine internals and will use database field default (empty dict) to populate to new form or to not filled field in existing form.

In the same time, we are allowing extending of this behaviour, and not creating field in database, if default value provided as `None`. In this case, generated field for new form will be empty, without any pre-filled value.

Same empty field will be displayed in case, when both `default=None` and `null=True` selected, during database form initialization. In this case form field will be empty, without any placeholder, but on save `null` object will be created in document.

Also, we even support separated defaults for form field and database field, allowing any form+database behaviour.

3.9.2 Examples

DictField with default empty dict value

Will place {} to form for existing/new fields. This value is hardcodded in parent MongoEngine project.

```
"""dict_demo.py"""
from example_app.models import db


class DictDemoModel(db.Document):
    """Documentation example model."""

    dict_field = db.DictField()
```

DictField with default `None` value, ignored by database

Reminder: Such field is empty in form, and will not create anything in database if not filled.

```
"""dict_demo.py"""
from example_app.models import db


class DictDemoModel(db.Document):
    """Documentation example model."""

    no_dict_field = db.DictField(default=None)
```

DictField with default `None` value, saved to database

Reminder: Such field is empty in form, and will create `null` object in database if not filled.

```
"""dict_demo.py"""
from example_app.models import db


class DictDemoModel(db.Document):
    """Documentation example model."""

    null_dict_field = db.DictField(default=None, null=True)
```

DictField with pre-defined default dict

This value is pre-defined on database level. So behaviour of form and in-code creation of such objects will be the same - default dict will be saved to database, if nothing provided to form/instance. Form will be pre-filled with default dict.

```
"""dict_demo.py"""
from example_app.models import db


```

(continues on next page)

(continued from previous page)

```
def get_default_dict():
    """Example of default dict specification."""
    return {"alpha": 1, "text": "text", "float": 1.2}


class DictDemoModel(db.Document):
    """Documentation example model."""

    dict_default = db.DictField(default=get_default_dict)
```

DictField with pre-defined value and no document object creation on `Null`

This is a case when you do not want to create any record in database document, if user completely delete pre-filled value in new document form. Here we use different `null` and `default` values in form field generation and during database object generation.

```
"""dict_demo.py"""
from example_app.models import db


def get_default_dict():
    """Example of default dict specification."""
    return {"alpha": 1, "text": "text", "float": 1.2}


class DictDemoModel(db.Document):
    """Documentation example model."""

    no_dict_prefilled = db.DictField(
        default=None,
        null=False,
        wtf_options={"default": get_default_dict, "null": True},
    )
```

DictField with pre-defined default dict with `Null` fallback

This is very rare case, when some default value is given, meaning that this value will be populated to the field, but if completely deleted, than `Null` will be saved in database.

```
"""dict_demo.py"""
from example_app.models import db


def get_default_dict():
    """Example of default dict specification."""
    return {"alpha": 1, "text": "text", "float": 1.2}


class DictDemoModel(db.Document):
    """Documentation example model."""
```

(continues on next page)

(continued from previous page)

```
    null_dict_default = db.DictField(default=get_default_dict, null=True)
```

3.10 EmailField

- API: *db_fields.EmailField*
- Default form field class: *MongoEmailField*

3.10.1 Form generation behaviour

Unlike *StringField* WTForm class of the field is not adjusted by normal form generation sequence and always match *MongoEmailField*. All other adjustments, related to validators insert are work with EmailField in the same way, as in *StringField*.

Additional `Email` validator is also inserted to form field, to exclude unnecessary database request, if form data incorrect.

Field respect user's adjustments in `wtf_field_class` option of *db_fields.EmailField*. This will change form field display, but will not change inserted validators.

3.10.2 Examples

strings_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not required EmailField

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    url_field = db.EmailField()
```

Required EmailField

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    required_url_field = db.EmailField(required=True)
```

3.11 EmbeddedDocumentField

Not yet documented. Please help us with new pull request.

3.12 FileField

Not yet documented. Please help us with new pull request.

3.13 FloatField

Changed in version 2.0.0: Default form field class changed from: `wtforms.fields.FloatField` to *MongoFloatField*.

- API: *db_fields.FloatField*
- Default form field class: *MongoFloatField*

3.13.1 Form generation behaviour

For Mongo database *FloatField* special WTForm field was created. This field's behaviour is the same, as for `wtforms.fields.FloatField`, but the widget is replaced to `NumberInput`, this should make a look of generated form better. It is possible, that in some cases usage of base, `wtforms.fields.FloatField` can be required by form design. Both fields are completely compatible, and replace can be done with `wtf_field_class` db form parameter.

If database field definition has any of `min_value` or `max_value`, then `NumberRange` validator will be added to form field.

3.13.2 Examples

numbers_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not limited FloatField

```
"""numbers_demo.py"""
from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    float_field_unlimited = db.FloatField()
```

Limited FloatField

```
"""numbers_demo.py"""
from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    float_field_limited = db.FloatField(min_value=float(1), max_value=200.455)
```

3.14 IntField

- API: *db_fields.IntField*
- Default form field class: wtforms.fields.IntegerField

3.14.1 Form generation behaviour

From form generation side this field is pretty standard and do not use any form generation adjustments.

If database field definition has any of `min_value` or `max_value`, then NumberRange validator will be added to form field.

3.14.2 Examples

numbers_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not limited IntField

```
"""numbers_demo.py"""
from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    integer_field_unlimited = db.IntField()
```

Limited IntField

```
"""numbers_demo.py"""
from example_app.models import db


class NumbersDemoModel(db.Document):
    """Documentation example model."""

    integer_field_limited = db.IntField(min_value=1, max_value=200)
```

3.15 ListField

Not yet documented. Please help us with new pull request.

3.16 ReferenceField

Not yet documented. Please help us with new pull request.

3.17 SortedListField (partly?)

Not yet documented. Please help us with new pull request.

3.18 StringField

- API: *db_fields.StringField*
- Default form field class: Selected by field settings combination

3.18.1 Form generation behaviour

By default, during WTForm generation for fields without specified size (`min_length` or `max_length`) class *MongoTextAreaField* is used, in case when `min_length` or `max_length` set, then *MongoStringField* used and `Length` will be added to form field validators. This allows to keep documents of any size in mongodb.

In some cases class *MongoStringField* is not the best choice for field, even with limited size. In this case user can easily overwrite generated field class by providing `wtf_field_class` on *db_fields.StringField* field declaration, as on document, as well as on form generation steps.

If database field definition has `regex` parameter set, then `Regexp` validator will be added to the form field.

3.18.2 Features deprecated

Field declaration step keyword arguments `password` and `textarea` are deprecated in Flask-Mongoengine version **2.0.0** and exist only to make migration steps easy.

To implement same behaviour, user should use `wtf_field_class` setting on *db_fields.StringField* init.

3.18.3 Related WTForm custom fields

Several special WTForms field implementation was created to support mongodb database behaviour and do not create any values in database, in case of empty fields. They can be used as `wtf_field_class` setting or independently. Some of them used in another database fields too, but all of them based on `wtforms.fields.StringField` and *EmptyStringIsNoneMixin*. You can use *EmptyStringIsNoneMixin* for own field types.

- *MongoEmailField*
- *MongoHiddenField*
- *MongoPasswordField*
- *MongoSearchField*
- *MongoStringField*
- *MongoTelField*
- *MongoTextAreaField*
- *MongoURLField*

3.18.4 Examples

strings_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not limited StringField as MongoTextAreaField

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    string_field = db.StringField()
```

Not limited StringField as MongoTelField

```
"""strings_demo.py"""
from example_app.models import db
from flask_mongoengine.wtf import fields as mongo_fields


class StringsDemoModel(db.Document):
    """Documentation example model."""

    tel_field = db.StringField(wtf_field_class=mongo_fields.MongoTelField)
```

Not limited StringField as MongoTextAreaField with https regex

mongoengine and wtforms projects are not consistent in how they work with regex. You will be safe, if you use `re.compile()` each time, when you work with regex settings, before parent projects itself.

```
"""strings_demo.py"""
import re

from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    regexp_string_field = db.StringField(regex=re.compile(
        r"^(https:\/\/)[\w.-]+(?:\.[\w\.-]+)+[\w\-\._~:/?#[\]@!\$&'\(\)\*\+,;=]+$"
    ))
```

Size limited StringField as MongoStringField

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    sized_string_field = db.StringField(min_length=5)
```

Required password field with minimum size

```
"""strings_demo.py"""
from example_app.models import db
from flask_mongoengine.wtf import fields as mongo_fields


class StringsDemoModel(db.Document):
    """Documentation example model."""

    password_field = db.StringField(
        wtf_field_class=mongo_fields.MongoPasswordField,
        required=True,
        min_length=5,
    )
```

3.19 URLField

- API: *db_fields.URLField*
- Default form field class: *MongoURLField*

3.19.1 Form generation behaviour

Unlike *StringField* WTForm class of the field is not adjusted by normal form generation sequence and always match *MongoURLField*. All other adjustments, related to validators insert are work with EmailField in the same way, as in *StringField*.

Additional `Regexp` validator is also inserted to form field, to exclude unnecessary database request, if form data incorrect. This validator use regexp, provided in `url_regex` of *db_fields.URLField*, or default URL regexp from mongoengine project. This is different from Flask-Mongoengine version **1.0.0** or earlier, where `URL` was inserted. This was changed, to exclude validators conflicts.

Important: *model_form()* is still use `URL` for compatibility with old setups.

Field respect user's adjustments in `wtf_field_class` option of *db_fields.URLField*. This will change form field display, but will not change inserted validators.

3.19.2 Examples

strings_demo.py in example app contain basic non-requirement example. You can adjust it to any provided example for test purposes.

Not required URLField

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    url_field = db.URLField()
```

Required URLField with minimum size

```
"""strings_demo.py"""
from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    required_url_field = db.URLField(required=True, min_length=25)
```

URLField with https only regexp check, if data exist

Regexp for `url_regex` should be prepared by `re`.

```
"""strings_demo.py"""
import re

from example_app.models import db


class StringsDemoModel(db.Document):
    """Documentation example model."""

    https_url_field = db.URLField(
        url_regex=re.compile(
            r"^(https:\/\/)[\w.-]+(?:\.[\w\.-]+)+[\w\-\._~:/?#[\]@!\$&'\(\)\*\+,;=]+$"
        ),
    )
```

3.20 Unsupported fields

3.20.1 CachedReferenceField

Not yet documented. Please help us with new pull request.

3.20.2 DynamicField

Not yet documented. Please help us with new pull request.

3.20.3 EmbeddedDocumentListField

Not yet documented. Please help us with new pull request.

3.20.4 EnumField

Not yet documented. Please help us with new pull request.

3.20.5 GenericEmbeddedDocumentField

Not yet documented. Please help us with new pull request.

3.20.6 GenericLazyReferenceField

Not yet documented. Please help us with new pull request.

3.20.7 GeoJsonBaseField

Not yet documented. Please help us with new pull request.

3.20.8 GeoPointField

Not yet documented. Please help us with new pull request.

3.20.9 ImageField

Not yet documented. Please help us with new pull request.

3.20.10 LazyReferenceField

Not yet documented. Please help us with new pull request.

3.20.11 LineStringField

Not yet documented. Please help us with new pull request.

3.20.12 LongField

Not yet documented. Please help us with new pull request.

3.20.13 MapField

Not yet documented. Please help us with new pull request.

3.20.14 MultiLineStringField

Not yet documented. Please help us with new pull request.

3.20.15 MultiPointField

Not yet documented. Please help us with new pull request.

3.20.16 MultiPolygonField

Not yet documented. Please help us with new pull request.

3.20.17 PointField

Not yet documented. Please help us with new pull request.

3.20.18 PolygonField

Not yet documented. Please help us with new pull request.

3.20.19 SequenceField

Not yet documented. Please help us with new pull request.

3.20.20 UUIDField

Not yet documented. Please help us with new pull request.

3.21 Unsure

3.21.1 GenericReferenceField

Not yet documented. Please help us with new pull request.

3.21.2 ObjectIdField

Not yet documented. Please help us with new pull request.

CHAPTER

FOUR

MIGRATION TO 2.0.0 AND CHANGES

4.1 Empty fields are not created in database

If you are already aware about empty string vs `None` database conflicts, you should know that some string based generated forms behaviour are not consistent between version **<=1.0.0** and **2.0.0+**.

In version **2.0.0** all fields with empty strings will be considered as `None` and will not be saved by default (this is easy to change for particular field to keep old behaviour for existed projects).

This behaviour is different from previous versions, where *`StringField`* consider empty string as valid value, and save it to database. This is the opposite behavior against original mongoengine project and related database.

> **Warning:** To keep easy migration and correct deprecation steps *`model_form()`* and *`model_fields()`* still keep old behaviour.

To be completely clear, let look on example. Let make a completely meaningless model, with all field types from mongoengine, like this:

```
"""example_app.models.py"""
from flask_mongoengine import MongoEngine

db = MongoEngine()


class Todo(db.Document):
    """Test model for AllFieldsModel."""
    string = db.StringField()


class Embedded(db.EmbeddedDocument):
    """Test embedded for AllFieldsModel."""
    string = db.StringField()


class AllFieldsModel(db.Document):
    """Meaningless Document with all field types."""
    binary_field = db.BinaryField()
    boolean_field = db.BooleanField()
    date_field = db.DateField()
    date_time_field = db.DateTimeField()
```

(continues on next page)

(continued from previous page)

```
    decimal_field = db.DecimalField()
    dict_field = db.DictField()
    email_field = db.EmailField()
    embedded_document_field = db.EmbeddedDocumentField(document_type=Embedded)
    file_field = db.FileField()
    float_field = db.FloatField()
    int_field = db.IntField()
    list_field = db.ListField(field=db.StringField)
    reference_field = db.ReferenceField(document_type=Todo)
    sorted_list_field = db.SortedListField(field=db.StringField)
    string_field = db.StringField()
    url_field = db.URLField()
    cached_reference_field = db.CachedReferenceField(document_type=Todo)
    complex_date_time_field = db.ComplexDateTimeField()
    dynamic_field = db.DynamicField()
    embedded_document_list_field = db.EmbeddedDocumentListField(document_type=Embedded)
    enum_field = db.EnumField(enum=[1, 2])
    generic_embedded_document_field = db.GenericEmbeddedDocumentField()
    generic_lazy_reference_field = db.GenericLazyReferenceField()
    geo_json_base_field = db.GeoJsonBaseField()
    geo_point_field = db.GeoPointField()
    image_field = db.ImageField()
    lazy_reference_field = db.LazyReferenceField(document_type=Todo)
    line_string_field = db.LineStringField()
    long_field = db.LongField()
    map_field = db.MapField(field=db.StringField())
    multi_line_string_field = db.MultiLineStringField()
    multi_point_field = db.MultiPointField()
    multi_polygon_field = db.MultiPolygonField()
    point_field = db.PointField()
    polygon_field = db.PolygonField()
    sequence_field = db.SequenceField()
    uuid_field = db.UUIDField()
    generic_reference_field = db.GenericReferenceField(document_type=Todo)
    object_id_field = db.ObjectIdField()
```

Now let's make an example instance of such object and save it to db.

```
from example_app.models import AllFieldsModel

obj = AllFieldsModel()
obj.save()
```

On database side this document will be created:

```
{
    "_id": {
        "$oid": "62df93ac3fe82c8656aae60d"
    },
    "dict_field": {},
    "list_field": [],
    "sorted_list_field": [],
```

(continues on next page)

(continued from previous page)

```
    "embedded_document_list_field": [],
    "map_field": {},
    "sequence_field": 1
}
```

For empty form Flask-Mongoengine **2.0.0** will create exactly the same document, old project version will try to fill some fields, based on *StringField*.

CHAPTER

FIVE

CUSTOM QUERYSET

flask-mongoengine attaches the following methods to Mongoengine's default QuerySet:

- **get_or_404**: works like .get(), but calls abort(404) if the object DoesNotExist. Optional arguments: *message* - custom message to display.
- **first_or_404**: same as above, except for .first(). Optional arguments: *message* - custom message to display.
- **paginate**: paginates the QuerySet. Takes two arguments, *page* and *per_page*.
- **paginate_field**: paginates a field from one document in the QuerySet. Arguments: *field_name*, *doc_id*, *page*, *per_page*.

Examples:

```
# 404 if object doesn't exist
def view_todo(todo_id):
    todo = Todo.objects.get_or_404(_id=todo_id)

# Paginate through todo
def view_todos(page=1):
    paginated_todos = Todo.objects.paginate(page=page, per_page=10)

# Paginate through tags of todo
def view_todo_tags(todo_id, page=1):
    todo = Todo.objects.get_or_404(_id=todo_id)
    paginated_tags = todo.paginate_field('tags', page, per_page=10)
```

Properties of the pagination object include: iter_pages, next, prev, has_next, has_prev, next_num, prev_num.

In the template:

```
{# Display a page of todos #}
<ul>
    {% for todo in paginated_todos.items %}
        <li>{{ todo.title }}</li>
    {% endfor %}
</ul>

{# Macro for creating navigation links #}
{% macro render_navigation(pagination, endpoint) %}
  <div class=pagination>
  {% for page in pagination.iter_pages() %}
    {% if page %}
      {% if page != pagination.page %}
```

(continues on next page)

(continued from previous page)

```
        <a href="{{ url_for(endpoint, page=page) }}">{{ page }}</a>
      {% else %}
        <strong>{{ page }}</strong>
      {% endif %}
    {% else %}
      <span class=ellipsis>...</span>
    {% endif %}
  {% endfor %}
  </div>
{% endmacro %}

{{ render_navigation(paginated_todos, 'view_todos') }}
```

CHAPTER

SIX

MONGOENGINE AND WTFORMS

flask-mongoengine automatically generates WTForms from MongoEngine models:

```
from flask_mongoengine.wtf import model_form

class User(db.Document):
    email = db.StringField(required=True)
    first_name = db.StringField(max_length=50)
    last_name = db.StringField(max_length=50)

class Content(db.EmbeddedDocument):
    text = db.StringField()
    lang = db.StringField(max_length=3)

class Post(db.Document):
    title = db.StringField(max_length=120, required=True, validators=[validators.
↪InputRequired(message='Missing title.'),])
    author = db.ReferenceField(User)
    tags = db.ListField(db.StringField(max_length=30))
    content = db.EmbeddedDocumentField(Content)

PostForm = model_form(Post)

def add_post(request):
    form = PostForm(request.POST)
    if request.method == 'POST' and form.validate():
        # do something
        redirect('done')
    return render_template('add_post.html', form=form)
```

For each MongoEngine field, the most appropriate WTForm field is used. Parameters allow the user to provide hints if the conversion is not implicit::

```
PostForm = model_form(Post, field_args={'title': {'textarea': True}})
```

Supported parameters:

For fields with `choices`:

- `multiple` to use a SelectMultipleField
- `radio` to use a RadioField

For `StringField`:

- `password` to use a PasswordField
- `textarea` to use a TextAreaField

For `ListField`:

- `min_entries` to set the minimal number of entries

(By default, a StringField is converted into a TextAreaField if and only if it has no max_length.)

6.1 Supported fields

- StringField
- BinaryField
- URLField
- EmailField
- IntField
- FloatField
- DecimalField
- BooleanField
- DateTimeField
- **ListField** (using wtforms.fields.FieldList)
- SortedListField (duplicate ListField)
- **EmbeddedDocumentField** (using wtforms.fields.FormField and generating inline Form)
- **ReferenceField** (using wtforms.fields.SelectFieldBase with options loaded from QuerySet or Document)
- DictField

6.2 Not currently supported field types:

- ObjectIdField
- GeoLocationField
- GenericReferenceField

CHAPTER

SEVEN

SESSION INTERFACE

To use MongoEngine as your session store simple configure the session interface:

```
from flask_mongoengine import MongoEngine, MongoEngineSessionInterface

app = Flask(__name__)
db = MongoEngine(app)
app.session_interface = MongoEngineSessionInterface(db)
```

CHAPTER

EIGHT

MONGO DEBUG TOOLBAR PANEL

Changed in version 2.0.0.

Mongo Debug Toolbar Panel was completely rewritten in version **2.0.0**. Before this version Mongo Debug Toolbar Panel used patching of main pymongo create/update/delete methods. This was not the best approach, as raised some compatibility issues during pymongo project updates. Since version **2.0.0** we use pymongo monitoring functions to track requests and timings. This approach completely removes patching of external packages and more compatibility friendly.

New approach require some user side actions to proper panel installation. This is done to exclude 'silent' registration of event loggers, to prevent performance degradation in external projects and in projects, that do not require Mongo Debug Toolbar Panel functional.

Described approach change brings several side effects, that user should be aware of:

1. Now Mongo Debug Toolbar Panel shows real database time, as reported by database engine. Excluding time required for data delivery from database instance to Flask instance.
2. Mongo Debug Toolbar Panel do not display and track tracebacks anymore. It only monitors database requests. Nothing more.
3. Mongo Debug Toolbar Panel do not do anything, if monitoring engine not registered before Flask application connection setup. Monitoring listener should be registered before first database connection. This is external requirement.
4. Mongo Debug Toolbar Panel code is now covered by internal tests, raising overall project code quality.
5. Mongo Debug Toolbar Panel can work without any usage of other `flask_mongoengine` functions.
6. Mongo Debug Toolbar Panel do not split requests types anymore, this is because now it handle any requests, including aggregations, collection creating/deleting and any other, reported by pymongo monitoring. Making splitting of incomming events will bring high complexity to parsers, as there are a lot of mongoDb commmands exist.

8.1 Installation

To install and use Mongo Debug Toolbar Panel:

1. Add `'flask_mongoengine.panels.MongoDebugPanel'` to `DEBUG_TB_PANELS` of Flask Debug Toolbar.
2. Import `mongo_command_logger` in your Flask application initialization file.
3. Import `monitoring` from `pymongo` package in your Flask application initialization file.
4. Register `mongo_command_logger` as event listener in `pymongo`.
5. Init Flask app instance.

Example:

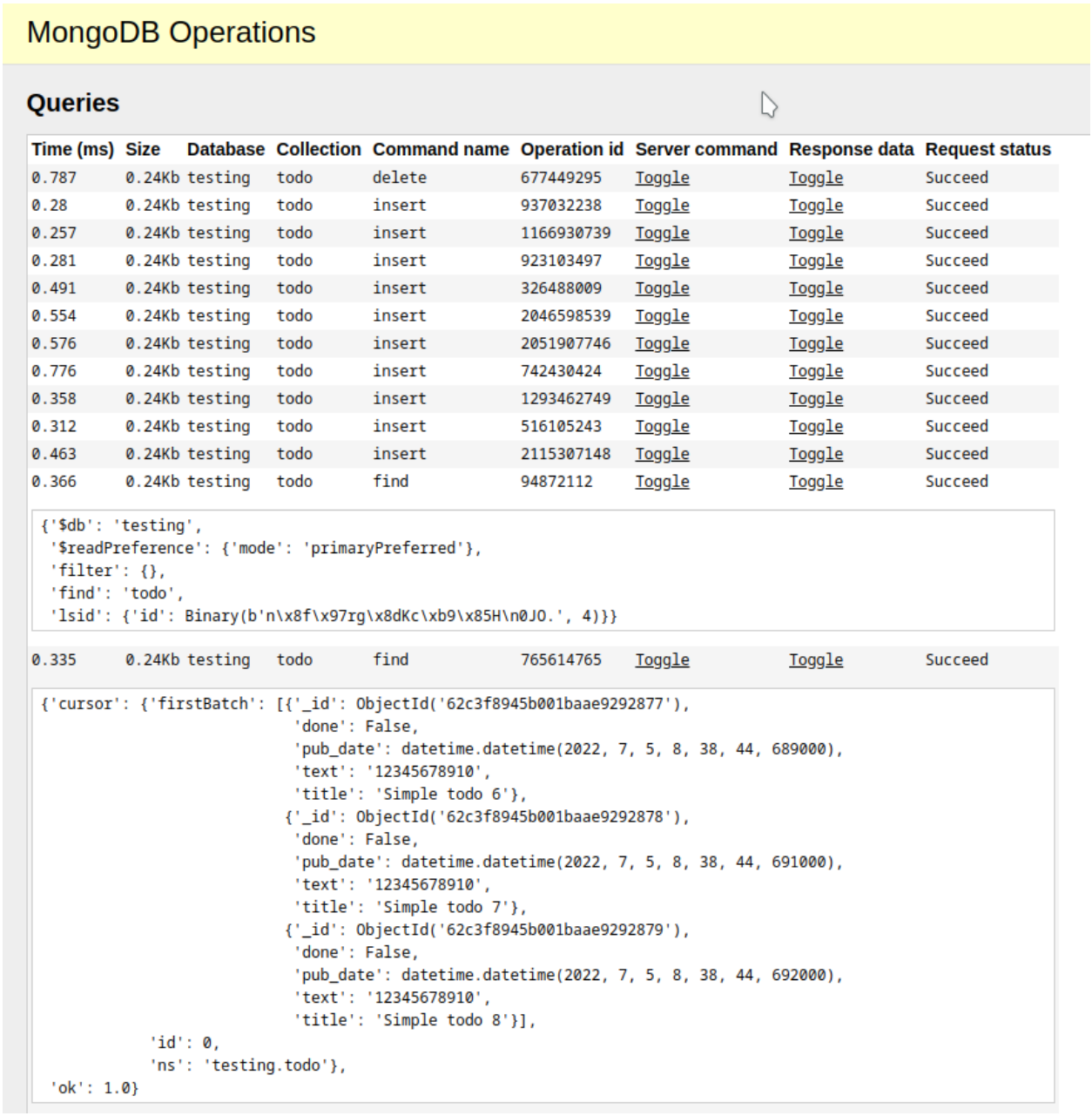
```
import flask
from flask_debugtoolbar import DebugToolbarExtension
from pymongo import monitoring

from flask_mongoengine.panels import mongo_command_logger
from flask_mongoengine import MongoEngine


app = flask.Flask(__name__)
app.config.from_object(__name__)
app.config["MONGODB_SETTINGS"] = {"DB": "testing", "host": "mongo"}
app.config["TESTING"] = True
app.config["SECRET_KEY"] = "some_key"
app.debug = True
app.config["DEBUG_TB_PANELS"] = ("flask_mongoengine.panels.MongoDebugPanel",)
DebugToolbarExtension(app)
monitoring.register(mongo_command_logger)
db = MongoEngine()
db.init_app(app)
```

Note: Working toolbar installation code can be found and tested in example_app, shipped with project codebase.

8.2 Configuration

You can add `MONGO_DEBUG_PANEL_SLOW_QUERY_LIMIT` variable to flask application config, to set a limit for queries highlight in web interface. By default, 100 ms.

8.3 Usage



- Mongo Debug Toolbar Panel logs every mongoDb query, in executed order.
- You can expand `Server command` to check what command was send to server.
- You can expand `Response data` to check raw response from server side.

8.4 Known issues

There is some HTML rendering related issues, that I cannot fix, as do not work with front end at all. If you have a little HTML/CSS knowledge, please help.

- #469 Mongo Debug Toolbar Panel: Update HTML view to use wide screens
- Objects sizes may be incorrect, as calculated in python. This part is copied from previous version, and may be removed in the future, if there will be confirmation, that this size data completely incorrect.

CHAPTER

NINE

EXAMPLE APP

A simple multi file app - to help get you started.

Completely located in docker container. Require ports 8000, 27015 on host system to be opened (for development needs).

Usage:

1. From flask-mongoengine root folder just run `docker-compose up`.
2. Wait until dependencies downloaded and containers up and running.
3. Open `http://localhost:8000/` to check the app.
4. Hit `ctrl+c` in terminal window to stop docker-compose.
5. You can use this app for live flask-mongoengine development and testing.

Note: Example app files located in `example_app` directory.

CHAPTER

TEN

API DOCUMENTATION

10.1 Base module API

This is the flask_mongoengine main modules API documentation.

10.1.1 flask_mongoengine.connection module

Module responsible for connection setup.

flask_mongoengine.connection.**_get_name**(*setting_name*)

Return known pymongo setting name, or lower case name for unknown.

This problem discovered in issue #451. As mentioned there pymongo settings are not case-sensitive, but mongoengine use exact name of some settings for matching, overwriting pymongo behaviour.

This function address this issue, and potentially address cases when pymongo will become case-sensitive in some settings by same reasons as mongoengine done.

Based on pymongo 4.1.1 settings.

Return type
str

flask_mongoengine.connection.**_sanitize_settings**(*settings*)

Remove MONGODB_ prefix from dict values, to correct bypass to mongoengine.

Return type
dict

flask_mongoengine.connection.**create_connections**(*config*)

Given Flask application's config dict, extract relevant config vars out of it and establish MongoEngine connection(s) based on them.

flask_mongoengine.connection.**get_connection_settings**(*config*)

Given a config dict, return a sanitized dict of MongoDB connection settings that we can then use to establish connections. For new applications, settings should exist in a MONGODB_SETTINGS key, but for backward compatibility we also support several config keys prefixed by MONGODB_, e.g. MONGODB_HOST, MONGODB_PORT, etc.

Return type
List[dict]

10.1.2 flask_mongoengine.db_fields module

Responsible for mongoengine fields extension, if WTFForms integration used.

class flask_mongoengine.db_fields.**BinaryField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, BinaryField

Extends mongoengine.fields.BinaryField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of *BinaryField*

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**BooleanField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, BooleanField

Extends mongoengine.fields.BooleanField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of *MongoBooleanField*

class flask_mongoengine.db_fields.**CachedReferenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, CachedReferenceField

Extends mongoengine.fields.CachedReferenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**ComplexDateTimeField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, ComplexDateTimeField

Extends mongoengine.fields.ComplexDateTimeField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

Important: During WTForm generation this field uses wtforms.fields.DateTimeLocalField with milliseconds accuracy. Direct microseconds not supported by browsers for this type of field. If exact microseconds support required, please use wtforms.fields.DateTimeField with extended text format set. Examples available in example app.

This does not affect on in database accuracy.

DEFAULT_WTF_FIELD

alias of DateTimeLocalField

property wtf_generated_options: dict

Extend form date time field with milliseconds support.

Return type
dict

class flask_mongoengine.db_fields.**DateField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, DateField

Extends mongoengine.fields.DateField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of DateField

class flask_mongoengine.db_fields.**DateTimeField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, DateTimeField

Extends mongoengine.fields.DateTimeField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of DateTimeLocalField

property wtf_generated_options: dict

Extend form date time field with milliseconds support.

Return type
dict

class flask_mongoengine.db_fields.**DecimalField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, DecimalField

Extends mongoengine.fields.DecimalField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_CHOICES_COERCE

alias of Decimal

DEFAULT_WTF_FIELD

alias of DecimalField

property **wtf_generated_options**: dict

Extend form validators with wtforms.validators.NumberRange.

Return type
dict

class flask_mongoengine.db_fields.**DictField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, DictField

Extends mongoengine.fields.DictField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of *MongoDictField*

property **wtf_generated_options**: dict

Extends default field options with *null* bypass.

Return type
dict

class flask_mongoengine.db_fields.**DynamicField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, DynamicField

Extends mongoengine.fields.DynamicField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**EmailField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, EmailField

Extends mongoengine.fields.EmailField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

Changed in version 2.0.0: Default form field output changed from *NoneStringField* to *flask_mongoengine.wtf.fields.MongoEmailField*

DEFAULT_WTF_FIELD

alias of *MongoEmailField*

property **wtf_generated_options**: dict

Extend form validators with wtforms.validators.Email

Return type
dict

class flask_mongoengine.db_fields.**EmbeddedDocumentField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, EmbeddedDocumentField

Extends mongoengine.fields.EmbeddedDocumentField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of FormField

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**EmbeddedDocumentListField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, EmbeddedDocumentListField

Extends mongoengine.fields.EmbeddedDocumentListField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**EnumField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, EnumField

Extends mongoengine.fields.EnumField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**FileField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, FileField

Extends mongoengine.fields.FileField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of FileField

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**FloatField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, FloatField

Extends mongoengine.fields.FloatField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

Changed in version 2.0.0: Default form field output changed from wtforms.fields.FloatField to *flask_mongoengine.wtf.fields.MongoFloatField* with 'numbers' input type.

DEFAULT_WTF_CHOICES_COERCE

alias of float

DEFAULT_WTF_FIELD

alias of *MongoFloatField*

property wtf_generated_options: dict

Extend form validators with wtforms.validators.NumberRange.

Return type
dict

class flask_mongoengine.db_fields.**GenericEmbeddedDocumentField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, GenericEmbeddedDocumentField

Extends mongoengine.fields.GenericEmbeddedDocumentField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**GenericLazyReferenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, GenericLazyReferenceField

Extends mongoengine.fields.GenericLazyReferenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**GenericReferenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, GenericReferenceField

Extends mongoengine.fields.GenericReferenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**GeoJsonBaseField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, GeoJsonBaseField

Extends mongoengine.fields.GeoJsonBaseField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**GeoPointField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, GeoPointField

Extends mongoengine.fields.GeoPointField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**ImageField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, ImageField

Extends mongoengine.fields.ImageField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**IntField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, IntField

Extends mongoengine.fields.IntField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_CHOICES_COERCE

alias of int

DEFAULT_WTF_FIELD

alias of IntegerField

property wtf_generated_options: dict

Extend form validators with wtforms.validators.NumberRange.

Return type
dict

class flask_mongoengine.db_fields.**LazyReferenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, LazyReferenceField

Extends mongoengine.fields.LazyReferenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**LineStringField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, LineStringField

Extends mongoengine.fields.LineStringField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**ListField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, ListField

Extends mongoengine.fields.ListField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of FieldList

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**LongField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, LongField

Extends mongoengine.fields.LongField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**MapField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, MapField

Extends mongoengine.fields.MapField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**MultiLineStringField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, MultiLineStringField

Extends mongoengine.fields.MultiLineStringField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**MultiPointField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, MultiPointField

Extends mongoengine.fields.MultiPointField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**MultiPolygonField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, MultiPolygonField

Extends mongoengine.fields.MultiPolygonField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**ObjectIdField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, ObjectIdField

Extends mongoengine.fields.ObjectIdField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_CHOICES_COERCE

alias of ObjectId

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**PointField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, PointField

Extends mongoengine.fields.PointField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**PolygonField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, PolygonField

Extends mongoengine.fields.PolygonField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**ReferenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, ReferenceField

Extends mongoengine.fields.ReferenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of *ModelSelectField*

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**SequenceField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, SequenceField

Extends mongoengine.fields.SequenceField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**SortedListField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, SortedListField

Extends mongoengine.fields.SortedListField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

DEFAULT_WTF_FIELD

alias of FieldList

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises

NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**StringField**(*, *password=False*, *textarea=False*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, StringField

Extends mongoengine.fields.StringField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

Changed in version 2.0.0: Default form field output changed from *NoneStringField* to *flask_mongoengine.wtf.fields.MongoTextAreaField*

DEFAULT_WTF_FIELD

alias of *MongoTextAreaField*

__init__(*, *password=False*, *textarea=False*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Extended *__init__()* method for mongoengine db field with WTForms options.

Parameters

- **password** (bool) – DEPRECATED: Force to use *MongoPasswordField* for field generation. In case of `password` and *wtf_field_class* both set, then *wtf_field_class* will be used.
- **textarea** (bool) – DEPRECATED: Force to use *MongoTextAreaField* for field generation. In case of `textarea` and *wtf_field_class* both set, then *wtf_field_class* will be used.
- **filters** (Union[List, Callable, None]) – DEPRECATED: wtf form field filters.
- **validators** (Union[List, Callable, None]) – DEPRECATED: wtf form field validators.
- **wtf_field_class** (Optional[Type]) – Any subclass of `wtforms.forms.core.Field` that can be used for form field generation. Takes precedence over *DEFAULT_WTF_FIELD* and `DEFAULT_WTF_CHOICES_FIELD`
- **wtf_filters** (Union[List, Callable, None]) – wtf form field filters.
- **wtf_validators** (Union[List, Callable, None]) – wtf form field validators.
- **wtf_choices_coerce** (Optional[Callable]) – Callable function to replace `DEFAULT_WTF_CHOICES_COERCE` for choices fields.

- **wtf_options** (Optional[dict]) – Dictionary with WTForm Field settings. Applied last, takes precedence over any generated field options.
- **kwargs** – keyword arguments silently bypassed to normal mongoengine fields

property wtf_field_class: Type

Parent class overwrite with support of class adjustment by field size.

Return type
Type

property wtf_generated_options: dict

Extend form validators with wtforms.validators.Regexp and wtforms.validators.Length.

Return type
dict

class flask_mongoengine.db_fields.**URLField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, URLField

Extends mongoengine.fields.URLField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

Changed in version 2.0.0: Default form field output changed from *NoneStringField* to *MongoURLField*

Changed in version 2.0.0: Now appends Regexp and use regexp provided to __init__ `url_regex`, instead of using non-configurable regexp from URL. This includes configuration conflicts, between modules.

DEFAULT_WTF_FIELD

alias of *MongoURLField*

property wtf_generated_options: dict

Extend form validators with wtforms.validators.Regexp

Return type
dict

class flask_mongoengine.db_fields.**UUIDField**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: *WtfFieldMixin*, UUIDField

Extends mongoengine.fields.UUIDField with wtf required parameters.

For full list of arguments and keyword arguments, look parent field docs. All arguments should be passed as keyword arguments, to exclude unexpected behaviour.

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Protection from execution of *to_wtf_field()* in form generation.

Raises
NotImplementedError – Field converter to WTForm Field not implemented.

class flask_mongoengine.db_fields.**WtfFieldMixin**(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Bases: object

Extension wrapper class for mongoengine BaseField.

This enables flask-mongoengine wtf to extend the number of field parameters, and settings on behalf of document model form generator for WTForm.

Class variables:

Variables

- **DEFAULT_WTF_CHOICES_FIELD** – Default WTForms Field used for db fields when **choices** option specified.
- **DEFAULT_WTF_FIELD** – Default WTForms Field used for db field.

DEFAULT_WTF_CHOICES_COERCE

alias of str

DEFAULT_WTF_CHOICES_FIELD

alias of SelectField

DEFAULT_WTF_FIELD = None

__init__(*, *validators=None*, *filters=None*, *wtf_field_class=None*, *wtf_filters=None*, *wtf_validators=None*, *wtf_choices_coerce=None*, *wtf_options=None*, ***kwargs*)

Extended *__init__()* method for mongoengine db field with WTForms options.

Parameters

- **filters** (Union[List, Callable, None]) – DEPRECATED: wtf form field filters.
- **validators** (Union[List, Callable, None]) – DEPRECATED: wtf form field validators.
- **wtf_field_class** (Optional[Type]) – Any subclass of wtforms.forms.core.Field that can be used for form field generation. Takes precedence over *DEFAULT_WTF_FIELD* and *DEFAULT_WTF_CHOICES_FIELD*
- **wtf_filters** (Union[List, Callable, None]) – wtf form field filters.
- **wtf_validators** (Union[List, Callable, None]) – wtf form field validators.
- **wtf_choices_coerce** (Optional[Callable]) – Callable function to replace *DEFAULT_WTF_CHOICES_COERCE* for choices fields.
- **wtf_options** (Optional[dict]) – Dictionary with WTForm Field settings. Applied last, takes precedence over any generated field options.
- **kwargs** – keyword arguments silently bypassed to normal mongoengine fields

static _ensure_callable_or_list(*argument*, *msg_flag*)

Ensure submitted argument value is a callable object or valid list value.

Parameters

- **argument** – Argument input to make verification on.
- **msg_flag** (str) – Argument string name for error message.

Return type

Optional[List]

to_wtf_field(*, *model=None*, *field_kwargs=None*)

Default WTFFormField generator for most of the fields.

Parameters

- **model** (Optional[Type]) – Document of model from *documents*, passed by *to_wtf_form()* for field types with other Document type dependency signature compatibility.
- **field_kwargs** (Optional[dict]) – Final field generation adjustments, passed for custom Forms generation from *to_wtf_form()* `fields_kwargs` parameter.

property wtf_field_class: Type

Final WTForm Field class, that will be used for field generation.

Return type
Type

property wtf_field_options: dict

Final WTForm Field options that will be applied as *wtf_field_class* kwargs.

Can be overwritten by *to_wtf_field()* if *to_wtf_form()* called with related field name in `fields_kwargs`.

It is not recommended to overwrite this property, for logic update overwrite *wtf_generated_options*

Return type
dict

property wtf_generated_options: dict

WTForm Field options generated by class, not updated by user provided `wtf_options`.

Return type
dict

flask_mongoengine.db_fields.**_setup_numbers_common_validators**(*options*, *obj*)

Extend `base_options` with common validators for number types.

Parameters

- **options** (dict) – dict, usually from *WtfFieldMixin.wtf_generated_options*
- **obj** (Union[IntField, DecimalField, FloatField]) – Any `mongoengine.fields.IntField` or `mongoengine.fields.DecimalField` or `mongoengine.fields.FloatField` subclasses instance.

Return type
dict

flask_mongoengine.db_fields.**_setup_strings_common_validators**(*options*, *obj*)

Extend `base_options` with common validators for string types.

Parameters

- **options** (dict) – dict, usually from *WtfFieldMixin.wtf_generated_options*
- **obj** (StringField) – Any `mongoengine.fields.StringField` subclass instance.

Return type
dict

10.1.3 flask_mongoengine.decorators module

Collection of project wide decorators.

`flask_mongoengine.decorators.`**orm_deprecated**(*func*)

Warning about usage of deprecated functions, that will be removed in the future.

`flask_mongoengine.decorators.`**wtf_required**(*func*)

Special decorator to warn user on incorrect installation.

10.1.4 flask_mongoengine.documents module

Extended version of `mongoengine.document`.

class `flask_mongoengine.documents.`**BaseQuerySet**(*document*, *collection*)

Bases: `QuerySet`

Extends `QuerySet` class with handly methods.

_abort_404(*_message_404*)

Returns 404 error with message, if message provided.

Parameters
_message_404 – Message for 404 comment

first_or_404(*_message_404=None*)

Same as *get_or_404()*, but uses `first()`, not `get()`.

Parameters
_message_404 – Message for 404 comment, not forwarded to `get()`

get_or_404(**args*, *_message_404=None*, ***kwargs*)

Get a document and raise a 404 Not Found error if it doesn't exist.

Parameters

- **_message_404** – Message for 404 comment, not forwarded to `get()`
- **args** – args list, silently forwarded to `get()`
- **kwargs** – keywords arguments, silently forwarded to `get()`

paginate(*page*, *per_page*)

Paginate the QuerySet with a certain number of docs per page and return docs for a given page.

paginate_field(*field_name*, *doc_id*, *page*, *per_page*, *total=None*)

Paginate items within a list field from one document in the QuerySet.

class `flask_mongoengine.documents.`**Document**(**args*, ***values*)

Bases: *WtfFormMixin*, `Document`

Abstract Document with QuerySet and WTForms extra helpers.

__objects

_cached_reference_fields = []

_class_name = 'Document'

_collection = None

_db_field_map = {}

_fields = {}

_fields_ordered = ()

_is_base_cls = False

_is_document = True

_meta = {'abstract': True, 'queryset_class': <class 'flask_mongoengine.documents.BaseQuerySet'>}

_reverse_db_field_map = {}

_subclasses = ('Document',)

_superclasses = ()

_types = ('Document',)

paginate_field(*field_name*, *page*, *per_page*, *total=None*)

Paginate items within a list field.

class flask_mongoengine.documents.DynamicDocument(**args*, ***values*)

Bases: *WtfFormMixin*, DynamicDocument

Abstract DynamicDocument with QuerySet and WTForms extra helpers.

_cached_reference_fields = []

_class_name = 'DynamicDocument'

_collection = None

_db_field_map = {}

_fields = {}

_fields_ordered = ()

_is_base_cls = False

_is_document = True

_meta = {'abstract': True, 'queryset_class': <class 'flask_mongoengine.documents.BaseQuerySet'>}

_reverse_db_field_map = {}

_subclasses = ('DynamicDocument',)

_superclasses = ()

_types = ('DynamicDocument',)

class flask_mongoengine.documents.DynamicEmbeddedDocument(**args*, ***kwargs*)

Bases: *WtfFormMixin*, DynamicEmbeddedDocument

Abstract DynamicEmbeddedDocument document with extra WTForms helpers.

_cached_reference_fields = []

_class_name = 'DynamicEmbeddedDocument'

_db_field_map = {}

_fields = {}

_fields_ordered = ()

_instance

_is_document = False

_meta = {'abstract': True}

_reverse_db_field_map = {}

_subclasses = ('DynamicEmbeddedDocument',)

_superclasses = ()

_types = ('DynamicEmbeddedDocument',)

class flask_mongoengine.documents.**EmbeddedDocument**(**args*, ***kwargs*)

Bases: *WtfFormMixin*, EmbeddedDocument

Abstract EmbeddedDocument document with extra WTForms helpers.

_cached_reference_fields = []

_class_name = 'EmbeddedDocument'

_db_field_map = {}

_fields = {}

_fields_ordered = ()

_instance

_is_document = False

_meta = {'abstract': True}

_reverse_db_field_map = {}

_subclasses = ('EmbeddedDocument',)

_superclasses = ()

_types = ('EmbeddedDocument',)

class flask_mongoengine.documents.**WtfFormMixin**

Bases: object

Special mixin, for form generation functions.

classmethod _get_fields_names(*only*, *exclude*)

Filter fields names for further form generation.

Parameters

- **only** (Optional[List[str]]) – An optional iterable with the property names that should be included in the form. Only these properties will have fields. Fields are always appear in provided order, this allows user to change form fields ordering, without changing database model.
- **exclude** (Optional[List[str]]) – An optional iterable with the property names that should be excluded from the form. All other properties will have fields. Fields are appears in order, defined in model, excluding provided fields names. For adjusting fields ordering, use `only`.

classmethod to_wtf_form(*cls*, *base_class=<class 'flask_mongoengine.wtf.models.ModelForm'>*, *only=None*, *exclude=None*, *fields_kwargs=None*)

Generate WTForm from Document model.

Parameters

- **base_class** (Type[*ModelForm*]) – Base form class to extend from. Must be a *ModelForm* subclass.
- **only** (Optional[List[str]]) – An optional iterable with the property names that should be included in the form. Only these properties will have fields. Fields are always appear in provided order, this allows user to change form fields ordering, without changing database model.
- **exclude** (Optional[List[str]]) – An optional iterable with the property names that should be excluded from the form. All other properties will have fields. Fields are appears in order, defined in model, excluding provided fields names. For adjusting fields ordering, use `only`.
- **fields_kwargs** (Optional[Dict[str, Dict]]) – An optional dictionary of dictionaries, where field names mapping to keyword arguments used to construct each field object. Has the highest priority over all fields settings (made in Document field definition). Field options are directly passed to field generation, so must match WTForm Field keyword arguments. Support special field keyword option `wtf_field_class`, that can be used for complete field class replacement.

  Dictionary format example:

```
dictionary = {
    "field_name":{
        "label":"new",
        "default": "new",
        "wtf_field_class": wtforms.fields.StringField
    }
}
```

  With such dictionary for field with name `field_name` `wtforms.fields.StringField` will be called like:

```
field_name = wtforms.fields.StringField(label="new", default="new")
```

Return type

Type[*ModelForm*]

10.1.5 flask_mongoengine.json module

Flask application JSON extension functions.

flask_mongoengine.json.**_convert_mongo_objects**(*obj*)

Convert objects, related to Mongo database to JSON.

flask_mongoengine.json.**_make_encoder**(*superclass*)

Extend Flask JSON Encoder 'default' method with support of Mongo objects.

flask_mongoengine.json.**_update_json_provider**(*superclass*)

Extend Flask Provider 'default' static method with support of Mongo objects.

flask_mongoengine.json.**override_json_encoder**(*app*)

A function to dynamically create a new MongoEngineJSONEncoder class based upon a custom base class. This function allows us to combine MongoEngine serialization with any changes to Flask's JSONEncoder which a user may have made prior to calling init_app.

NOTE: This does not cover situations where users override an instance's json_encoder after calling init_app.

flask_mongoengine.json.**use_json_provider**()

Split Flask before 2.2.0 and after, to use/not use JSON provider approach.

Return type
bool

10.1.6 flask_mongoengine.pagination module

Module responsible for custom pagination.

class flask_mongoengine.pagination.**ListFieldPagination**(*queryset*, *doc_id*, *field_name*, *page*, *per_page*, *total=None*)

Bases: *Pagination*

next(*error_out=False*)

Returns a *Pagination* object for the next page.

prev(*error_out=False*)

Returns a *Pagination* object for the previous page.

class flask_mongoengine.pagination.**Pagination**(*iterable*, *page*, *per_page*)

Bases: object

property has_next

True if a next page exists.

property has_prev

True if a previous page exists

iter_pages(*left_edge=2*, *left_current=2*, *right_current=5*, *right_edge=2*)

Iterates over the page numbers in the pagination. The four parameters control the thresholds how many numbers should be produced from the sides. Skipped page numbers are represented as *None*. This is how you could render such a pagination in the templates:

```
{% macro render_pagination(pagination, endpoint) %}
  <div class=pagination>
  {%- for page in pagination.iter_pages() %}
    {% if page %}
      {% if page != pagination.page %}
        <a href="{{ url_for(endpoint, page=page) }}">{{ page }}</a>
      {% else %}
        <strong>{{ page }}</strong>
      {% endif %}
    {% else %}
      <span class=ellipsis>...</span>
    {% endif %}
  {%- endfor %}
  </div>
{% endmacro %}
```

next(*error_out=False*)

Returns a *Pagination* object for the next page.

property next_num

Number of the next page

property pages

The total number of pages

prev(*error_out=False*)

Returns a *Pagination* object for the previous page.

property prev_num

Number of the previous page.

10.1.7 flask_mongoengine.panels module

Debug panel views and logic and related mongoDb event listeners.

class flask_mongoengine.panels.**RawQueryEvent**(*_event*, *_start_event*, *_is_query_pass*)

Bases: object

Responsible for parsing monitoring events to web panel interface.

Parameters

- **_event** (Union[CommandSucceededEvent, CommandFailedEvent]) – Succeeded or Failed event object from pymongo monitoring.
- **_start_event** (CommandStartedEvent) – Started event object from pymongo monitoring.
- **_is_query_pass** (bool) – Boolean status of db query reported by pymongo monitoring.

_event: Union[CommandSucceededEvent, CommandFailedEvent]

_start_event: CommandStartedEvent

_is_query_pass: bool

property time

Query execution time.

property size

Query object size.

property database

Query database target.

property collection

Query collection target.

property command_name

Query db level operation/command name.

property operation_id

MongoDb operation_id used to match 'start' and 'final' monitoring events.

property server_command

Raw MongoDb command send to server.

property server_response

Raw MongoDb response received from server.

property request_status

Query execution status.

class flask_mongoengine.panels.**MongoCommandLogger**

Bases: CommandListener

Receive point for CommandListener events.

Count and parse incoming events for display in debug panel.

append_raw_query(*event*, *request_status*)

Pass 'unknown' events to parser and include final result to final list.

failed(*event*)

Receives 'failed' events. Required to track database answer to request.

reset_tracker()

Resets all counters to default, keeping instance itself the same.

started(*event*)

Receives 'started' events. Required to track original request context.

succeeded(*event*)

Receives 'succeeded' events. Required to track database answer to request.

flask_mongoengine.panels.**_maybe_patch_jinja_loader**(*jinja_env*)

Extend jinja_env loader with flask_mongoengine templates folder.

class flask_mongoengine.panels.**MongoDebugPanel**(**args*, ***kwargs*)

Bases: DebugPanel

Panel that shows information about MongoDB operations.

config_error_message = 'Pymongo monitoring configuration error. mongo_command_logger should be registered before database connection.'

name = 'MongoDB'

has_content = True

property _context: dict

Context for rendering, as property for easy testing.

Return type
dict

property is_properly_configured: bool

Checks that all required watchers registered before Flask application init.

Return type
bool

process_request(*request*)

Resets logger stats between each request.

nav_title()

Debug toolbar in the bottom right corner.

Return type
str

nav_subtitle()

Count operations total time.

Return type
str

title()

Title for 'opened' debug panel window.

Return type
str

url()

Placeholder for internal URLs.

Return type
str

content()

Gathers all template required variables in one dict.

10.1.8 flask_mongoengine.sessions module

class flask_mongoengine.sessions.**MongoEngineSession**(*initial=None*, *sid=None*)

Bases: CallbackDict, SessionMixin

_abc_impl = <_abc._abc_data object>

class flask_mongoengine.sessions.**MongoEngineSessionInterface**(*db*, *collection='session'*)

Bases: SessionInterface

SessionInterface for mongoengine

get_expiration_time(*app*, *session*)

A helper method that returns an expiration date for the session or `None` if the session is linked to the browser session. The default implementation returns now + the permanent session lifetime configured on the application.

Return type
`timedelta`

open_session(*app*, *request*)

This is called at the beginning of each request, after pushing the request context, before matching the URL.

This must return an object which implements a dictionary-like interface as well as the `SessionMixin` interface.

This will return `None` to indicate that loading failed in some way that is not immediately an error. The request context will fall back to using `make_null_session()` in this case.

save_session(*app*, *session*, *response*)

This is called at the end of each request, after generating a response, before removing the request context. It is skipped if `is_null_session()` returns `True`.

10.1.9 Module contents

Note: Parent mongoengine project docs/docstrings has some formatting issues. If class/function/method link not clickable, search on provided parent documentation manually.

class flask_mongoengine.**MongoEngine**(*app=None*, *config=None*)

Bases: `object`

Main class used for initialization of Flask-MongoEngine.

property connection: `dict`

Return MongoDB connection(s) associated with this MongoEngine instance.

Return type
`dict`

init_app(*app*, *config=None*)

flask_mongoengine.**current_mongoengine_instance**()

Return a MongoEngine instance associated with current Flask app.

10.2 WTF module API

This is the flask_mongoengine.wtf modules API documentation.

10.2.1 flask_mongoengine.wtf.fields module

Useful form fields for use with the mongoengine.

class flask_mongoengine.wtf.fields.**BinaryField**(**args*, ***kwargs*)

Bases: TextAreaField

Custom TextAreaField that converts its value with binary_type.

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters
valuelist – A list of strings to process.

class flask_mongoengine.wtf.fields.**DictField**(**args*, ***kwargs*)

Bases: *JSONField*

Special version fo *JSONField* to be generated for *flask_mongoengine.db_fields.DictField*.

Used in generator before flask_mongoengine version 2.0

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters
valuelist – A list of strings to process.

class flask_mongoengine.wtf.fields.**EmptyStringIsNoneMixin**

Bases: object

Special mixin to ignore empty strings **before** parent class processing.

Unlike old *NoneStringField* we do it before parent class call, this allows us to reuse this mixin in many more cases without errors.

process_formdata(*valuelist*)

Ignores empty string and calls parent *process_formdata()* if data present.

Parameters
valuelist – A list of strings to process.

class flask_mongoengine.wtf.fields.**JSONField**(**args*, ***kwargs*)

Bases: TextAreaField

Special version fo wtforms.fields.TextAreaField.

_value()

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters
valuelist – A list of strings to process.

class flask_mongoengine.wtf.fields.**ModelSelectField**(**args*, ***kwargs*)

Bases: *QuerySetSelectField*

Like a QuerySetSelectField, except takes a model class instead of a queryset and lists everything in it.

__init__(*label=''*, *validators=None*, *model=None*, ***kwargs*)

Init docstring placeholder.

class flask_mongoengine.wtf.fields.**ModelSelectMultipleField**(**args*, ***kwargs*)

Bases: *QuerySetSelectMultipleField*

Allows multiple select

__init__(*label=''*, *validators=None*, *model=None*, ***kwargs*)

Init docstring placeholder.

class flask_mongoengine.wtf.fields.**MongoBooleanField**(**args*, ***kwargs*)

Bases: `SelectField`

Mongo SelectField field for BooleanFields, that correctly coerce values.

__init__(*label=None*, *validators=None*, *coerce=None*, *choices=None*, *validate_choice=True*, ***kwargs*)

Replaces defaults of `wtforms.fields.SelectField` with for Boolean values.

Fully compatible with `wtforms.fields.SelectField` and have same parameters.

class flask_mongoengine.wtf.fields.**MongoDictField**(*json_encoder=None*, *json_encoder_kwargs=None*, *json_decoder=None*, *json_decoder_kwargs=None*, *null=None*, **args*, ***kwargs*)

Bases: *MongoTextAreaField*

Form field to handle JSON in *DictField*.

__init__(*json_encoder=None*, *json_encoder_kwargs=None*, *json_decoder=None*, *json_decoder_kwargs=None*, *null=None*, **args*, ***kwargs*)

Special WTForms field for *DictField*

Configuration available with providing `wtf_options` on *DictField* initialization.

Parameters

- **json_encoder** (`Optional[Callable]`) – Any function, capable to transform dict to string, by default `json.dumps()`
- **json_encoder_kwargs** (`Optional[dict]`) – Any dictionary with parameters to `json_encoder()`, by default: *{"indent":4}*
- **json_decoder** (`Optional[Callable]`) – Any function, capable to transform string to dict, by default `json.loads()`
- **json_decoder_kwargs** (`Optional[dict]`) – Any dictionary with parameters to `json_decoder()`, by default: *{}*

_ensure_data_is_dict()

Ensures that saved data is dict, not a list or other valid parsed JSON.

_parse_json_data()

Tries to load JSON data with python internal JSON library.

_value()

Show existing data as pretty-formatted, or show raw data/empty dict.

process_formdata(*valuelist*)

Process text form data to dictionary or raise JSONDecodeError.

class flask_mongoengine.wtf.fields.**MongoEmailField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, EmailField

Regular wtforms.fields.EmailField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoFloatField**(**args*, ***kwargs*)

Bases: FloatField

Regular wtforms.fields.FloatField, with widget replaced to wtforms.widgets.NumberInput.

widget = <wtforms.widgets.core.NumberInput object>

class flask_mongoengine.wtf.fields.**MongoHiddenField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, HiddenField

Regular wtforms.fields.HiddenField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoPasswordField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, PasswordField

Regular wtforms.fields.PasswordField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoSearchField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, SearchField

Regular wtforms.fields.SearchField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoStringField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, StringField

Regular wtforms.fields.StringField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoTelField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, TelField

Regular wtforms.fields.TelField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoTextAreaField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, TextAreaField

Regular wtforms.fields.TextAreaField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**MongoURLField**(**args*, ***kwargs*)

Bases: *EmptyStringIsNoneMixin*, URLField

Regular wtforms.fields.URLField, that transform empty string to *None*.

class flask_mongoengine.wtf.fields.**NoneStringField**(**args*, ***kwargs*)

Bases: StringField

Custom StringField that counts “” as None

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters

valuelist – A list of strings to process.

class flask_mongoengine.wtf.fields.**QuerySetSelectField**(**args*, ***kwargs*)

Bases: SelectFieldBase

Given a QuerySet either at initialization or inside a view, will display a select drop-down field of choices. The *data* property actually will store/keep an ORM model instance, not the ID. Submitting a choice which is not in the queryset will result in a validation error.

Specifying *label_attr* in the constructor will use that property of the model instance for display in the list, else the model object's *__str__* or *__unicode__* will be used.

If *allow_blank* is set to *True*, then a blank choice will be added to the top of the list. Selecting this choice will result in the *data* property being *None*. The label for the blank choice can be set by specifying the *blank_text* parameter.

__init__(*label=''*, *validators=None*, *queryset=None*, *label_attr=''*, *allow_blank=False*, *blank_text='---'*, *label_modifier=None*, ***kwargs*)

Init docstring placeholder.

_is_selected(*item*)

iter_choices()

Provides data for choice widget rendering. Must return a sequence or iterable of (value, label, selected) tuples.

pre_validate(*form*)

Field-level validation. Runs before any other validators.

Parameters
form – The form the field belongs to.

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters
valuelist – A list of strings to process.

widget = <wtforms.widgets.core.Select object>

class flask_mongoengine.wtf.fields.**QuerySetSelectMultipleField**(**args*, ***kwargs*)

Bases: *QuerySetSelectField*

Same as *QuerySetSelectField* but with multiselect options.

__init__(*label=''*, *validators=None*, *queryset=None*, *label_attr=''*, *allow_blank=False*, *blank_text='---'*, ***kwargs*)

Init docstring placeholder.

_is_selected(*item*)

process_formdata(*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters
valuelist – A list of strings to process.

widget = <wtforms.widgets.core.Select object>

flask_mongoengine.wtf.fields.**coerce_boolean**(*value*)

Transform SelectField boolean value from string and in reverse direction.

Return type
Optional[bool]

10.2.2 flask_mongoengine.wtf.models module

class flask_mongoengine.wtf.models.**ModelForm**(**args*, ***kwargs*)

Bases: FlaskForm

A WTForms mongoengine model form

_unbound_fields = None

_wtforms_meta = None

model_class: Type[Union[Document, DynamicDocument]]

save(*commit=True*, ***kwargs*)

10.2.3 flask_mongoengine.wtf.orm module

Tools for generating forms based on mongoengine Document schemas.

class flask_mongoengine.wtf.orm.**ModelConverter**(*converters=None*)

Bases: object

_generate_convert_base_kwargs(*field*, *field_args*)

Return type
dict

classmethod _number_common(*model*, *field*, *kwargs*)

_process_convert_for_choice_fields(*field*, *field_class*, *kwargs*)

classmethod _string_common(*model*, *field*, *kwargs*)

coerce(*field_type*)

conv_Binary(*model*, *field*, *kwargs*)

conv_Boolean(*model*, *field*, *kwargs*)

conv_Date(*model*, *field*, *kwargs*)

conv_DateTime(*model*, *field*, *kwargs*)

conv_Decimal(*model*, *field*, *kwargs*)

conv_Dict(*model*, *field*, *kwargs*)

conv_Email(*model*, *field*, *kwargs*)

conv_EmbeddedDocument(*model*, *field*, *kwargs*)

conv_File(*model*, *field*, *kwargs*)

conv_Float(*model*, *field*, *kwargs*)

conv_GenericReference(*model*, *field*, *kwargs*)

conv_GeoLocation(*model*, *field*, *kwargs*)

conv_Int(*model*, *field*, *kwargs*)

conv_List(*model*, *field*, *kwargs*)

conv_ObjectId(*model*, *field*, *kwargs*)

conv_Reference(*model*, *field*, *kwargs*)

conv_SortedList(*model*, *field*, *kwargs*)

conv_String(*model*, *field*, *kwargs*)

conv_URL(*model*, *field*, *kwargs*)

convert(*model*, *field*, *field_args*)

flask_mongoengine.wtf.orm.**_get_fields_names**(*model*, *only*, *exclude*)

Filter fields names for further form generation.

Parameters

- **model** – Source model class for fields list retrieval
- **only** (Optional[List[str]]) – If provided, only these field names will have fields definition.
- **exclude** (Optional[List[str]]) – If provided, field names will be excluded from fields definition. All other field names will have fields.

Return type
List[str]

flask_mongoengine.wtf.orm.**converts**(**args*)

flask_mongoengine.wtf.orm.**model_fields**(*model*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*)

Generate a dictionary of fields for a given database model.

See *model_form()* docstring for description of parameters.

Return type
OrderedDict

flask_mongoengine.wtf.orm.**model_form**(*model*, *base_class=<class 'flask_mongoengine.wtf.models.ModelForm'>*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*)

Create a wtforms Form for a given mongoengine Document schema:

```
from flask_mongoengine.wtf import model_form
from myproject.myapp.schemas import Article
ArticleForm = model_form(Article)
```

Parameters

- **model** (Type[BaseDocument]) – A mongoengine Document schema class
- **base_class** (Type[*ModelForm*]) – Base form class to extend from. Must be a *ModelForm* subclass.
- **only** (Optional[List[str]]) – An optional iterable with the property names that should be included in the form. Only these properties will have fields. Fields are always appear in provided order, this allows user to change form fields ordering, without changing database model.
- **exclude** (Optional[List[str]]) – An optional iterable with the property names that should be excluded from the form. All other properties will have fields. Fields are appears in order, defined in model, excluding provided fields names. For adjusting fields ordering, use `only`.
- **field_args** – An optional dictionary of field names mapping to keyword arguments used to construct each field object.
- **converter** – A converter to generate the fields based on the model properties. If not set, *ModelConverter* is used.

Return type
Type[*ModelForm*]

10.2.4 Module contents

WTFForms integration module init file.

CHAPTER

ELEVEN

CONTRIBUTING GUIDE

MongoEngine has a large *community* and contributions are always encouraged. Contributions can be as simple as typo fix in the documentation, as well as complete new features and integrations. Please read these guidelines before sending a pull request.

11.1 Bugfixes and new features

Before starting to write code, look for existing tickets or create one for your specific issue or feature request. That way you avoid working on something that might not be of interest or that has already been addressed.

For new integrations do you best to make integration optional, to leave user opportunity to exclude additional dependencies, in case when user do not need integration or feature.

11.2 Supported interpreters

Flask-MongoEngine supports CPython 3.7 and newer, PyPy 3.7 and newer. Language features not supported by all interpreters can not be used.

11.3 Running tests

All development requirements, except docker are included in package extra options `dev`. So, to install full development environment you need just run package with all related options installation.

Our local test environment related on docker and nox to test project on real database engine and not use any database mocking, as such mocking can raise unexpected behaviour, that is not seen in real database engines.

Before running tests, please ensure that real database not launched on local port `27017`, otherwise tests will fail. If you want to run tests with local launched database engine, run tests in non-interactive mode (see below), in this case docker will not be used at all.

We do not include docker python package to requirements, to exclude harming local developer's system, if docker installed in recommended/other way.

To run minimum amount of required tests with docker, use:

```
nox
```

To run minimum amount of required tests with local database, use:

```
nox --non-interactive
```

To run one or mode nox sessions only, use `-s` option. For example to run only documentation and linting tests, run:

```
nox -s documentation_tests lint
```

In some cases you will want to bypass arguments to pytest itself, to run single test or single test file. It is easy to do, everything after double dash will be bypassed to pytest directly. For example, to run `test__normal_command__logged` test only, use:

```
nox -- -k test__normal_command__logged
```

11.4 Setting up the code for local development and tests

1. Fork the `flask-mongoengine` repo on GitHub.
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/flask-mongoengine.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development with all required development and testing dependencies (except docker):

   ```
   cd flask-mongoengine/
   # With all development and package requirements, except docker
   pip install -e .[wtf,toolbar,dev]
   ```

4. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests and lint check:

   ```
   nox
   ```

   Please note that nox runs lint and documentation builds check automatically, since we have a test environment for it. During lint check run, some common issues will be fixed automatically, so in case of failed status, firstly just try to rerun check again. If issue is not fixed automatically, check run log.

   If you feel like running only the lint environment, please use the following command:

   ```
   nox -s lint
   ```

6. Ensure that your feature or commit is fully covered by tests. Check report after regular nox run.
7. Please install pre-commit git hook before adding any commits. This will ensure/fix 95% of linting issues. Otherwise, GitHub CI/CD pipeline may fail. This is linting double check (same as in nox), but it will run always, even if you forget to run tests before commit.

   ```
   pre-commit install
   ```

8. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

9. Submit a pull request through the GitHub website.

11.5 Interactive documentation development

Our nox configuration include special session for simplifying documentation development. When launched, documentation will be re-rendered after any saved code/documentation files changes. To use this session, after local environment setup, just run:

```
nox -s docs
```

Rendered documentation will be available under: http://localhost:9812/

11.6 Style guide

11.6.1 General guidelines

- Avoid backward breaking changes if at all possible.
- Avoid mocking of external libraries; Mocking allowed only in tests.
- Avoid complex structures, for complex classes/methods/functions try to separate to little objects, to simplify code reuse and testing.
- Avoid code duplications, try to exctract duplicate code to function. (Code duplication in tests is allowed.)
- Write docstrings for new classes and methods.
- Write tests and make sure they pass.
- Add yourself to *AUTHORS.md* :)

11.6.2 Code style guide

- Docstrings are using RST format. Developer encouraged to include any signigicant information to docstrings.
- Developers are encouraged to include typing information in functions signatures.
- Developers should avoid including typing information to docstrings, if possible.
- Developers should avoid including typing information to docstrings and signatures in same objects, to avoid rendering conflicts.
- Code formatting is completely done by black and following black's style implementation of PEP8. Target python version is the lowest supported version.
- Code formatting should be done before passing any new merge requests.
- Code style should use f-strings if possible, exceptional can be made for expensive debug level logging statements.

11.6.3 Documentation style guide

- Documentation should use Markdown as main format. Including Rest syntax blocks inside are allowed. Exceptional is API auto modules documents. Our documentation engine support MyST markdown extensions.
- Documentation should use same 88 string length requirement, as code. Strings can be larger for cases, when last word/statement is any kind of link (either to class/function/attribute or web link).
- Weblinks should be placed at the end of document, to make documentation easy readable without rendering (in editor).
- Docs formatting should be checked and formatted with pre-commit plugin before submitting.

11.7 CI/CD testing

All tests are run on GitHub actions and any pull requests are automatically tested for full range of supported Flask, mongoengine, python and mongodb versions. Any pull requests without tests will take longer to be integrated and might be refused.

CHAPTER

TWELVE

CREDITS

12.1 Current maintainers:

- Andrey Shpak

This project is made by incredible authors:

- Adam Chainz
- aliaksandr.askerka
- Alistair Roche
- Almog Cohen
- Aly Sivji
- Andrew Elkins
- Andrey Shpak
- Anthony Nemitz
- Anton Antonov
- Axel Haustant
- bdadson
- bioneddy
- Boullier Jocelyn
- bright
- Bright Dadson
- BrightPan
- Bruno Belarmino
- Bruno Rocha
- Christian Wygoda
- Clay McClure
- Cory Dolphin
- Denny Huang
- Dragos Catarahia
- Emmanuel Leblond

- Fengda Huang
- fieliapm
- Garito
- Gregg Lind
- Ido Shraga
- icoz
- Jack Stouffer
- Jamar Parris
- jcass
- Jeff Tharp
- jmesquita
- Joao Mesquita
- Joe Shaw
- Joe Shaw
- Jorge Bastida
- jorgebastida
- JTG
- Jérôme Lafréchoux
- Jérôme Lafréchoux
- Len Buckens
- Liam McLoughlin
- losintikfos
- Lucas Nemeth
- Lucas Vogelsang
- Marcel Tschopp
- Marcus Carlsson
- Martin Hanzík
- Massimo Santini
- Mattias Granlund
- Max Countryman
- mickey06
- mxck
- Nauman Ahmad
- onlinejudge95
- Paul Brown
- Peter Kristensen

- PeterKharchenko
- Philip House
- qisoster
- rma4ok
- Rod Cloutier
- Ross Lawley
- Sebastian Karlsson
- Serge S. Koval
- Sibelius Seraphini
- Simon Eames
- Slavek Kabrda
- Stefan Wójcik
- Stephen Brown II
- Stormxx
- Thanathip Limna
- Thomas Steinacher
- Tim Gates
- Ty3uK
- vantastic
- wcdolphin
- woo
- yoyicue
- Zephor
-
-
-

This project inspired by two repos:

- danjac
- maratfm

CHAPTER

THIRTEEN

OLD CHANGELOG

13.1 Changes in 1.0.0

Changelog maintenance automated and latest changelog available at github release page.

Use version 0.9.5 if old dependencies required.

13.2 Changes in 0.9.5

- Disable flake8 on travis.
- Correct *Except* clauses in code.
- Fix warning about undefined unicode variable in orm.py with python 3

13.3 Changes in 0.9.4

- ADDED: Support for *MONGODB_CONNECT* mongodb parameter (#321)
- ADDED: Support for *MONGODB_TZ_AWARE* mongodb parameter.

13.4 Changes in 0.9.3

- Fix test and mongomock (#304)
- Run Travis builds in a container-based environment (#301)

13.5 Changes in 0.9.2

- Travis CI/CD pipeline update to automatically publish 0.9.1.

13.6 Changes in 0.9.1

- Fixed setup.py for various platforms (#298).
- Added Flask-WTF v0.14 support (#294).
- MongoEngine instance now holds a reference to a particular Flask app it was initialized with (#261).

13.7 Changes in 0.9.0

- BREAKING CHANGE: Dropped Python v2.6 support

13.8 Changes in 0.8.2

- Fixed relying on mongoengine.python_support.
- Fixed cleaning up empty connection settings #285

13.9 Changes in 0.8.1

- Fixed connection issues introduced in 0.8
- Removed support for MongoMock

13.10 Changes in 0.8

- Dropped MongoEngine 0.7 support
- Added MongoEngine 0.10 support
- Added PyMongo 3 support
- Added Python3 support up to 3.5
- Allowed empying value list in SelectMultipleField
- Fixed paginator issues
- Use InputRequired validator to allow 0 in required field
- Made help_text Field attribute optional
- Added “radio” form_arg to convert field into RadioField
- Added “textarea” form_arg to force conversion into TextAreaField
- Added field parameters (validators, filters…)
- Fixed ‘False’ connection settings ignored
- Fixed bug to allow multiple instances of extension
- Added MongoEngineSessionInterface support for PyMongo’s tz_aware option
- Support arbitrary primary key fields (not “id”)

- Configurable httponly flag for MongoEngineSessionInterface
- Various bugfixes, code cleanup and documentation improvements
- Move from deprecated flask.ext.* to flask_* syntax in imports
- Added independent connection handler for FlaskMongoEngine
- All MongoEngine connection calls are proxied via FlaskMongoEngine connection handler
- Added backward compatibility for settings key names
- Added support for MongoMock and temporary test DB
- Fixed issue with multiple DB support
- Various bugfixes

13.11 Changes in 0.7

- Fixed only / exclude in model forms (#49)
- Added automatic choices coerce for simple types (#34)
- Fixed EmailField and URLField rendering and validation (#44, #9)
- Use help_text for field description (#43)
- Fixed Pagination and added Document.paginate_field() helper
- Keep model_forms fields in order of creation
- Added MongoEngineSessionInterface (#5)
- Added customisation hooks for FieldList sub fields (#19)
- Handle non ascii chars in the MongoDebugPanel (#22)
- Fixed toolbar stacktrace if a html directory is in the path (#31)
- ModelForms no longer patch Document.update (#32)
- No longer wipe field kwargs in ListField (#20, #19)
- Passthrough ModelField.save-arguments (#26)
- QuerySetSelectMultipleField now supports initial value (#27)
- Clarified configuration documentation (#33)
- Fixed forms when EmbeddedDocument has no default (#36)
- Fixed multiselect restore bug (#37)
- Split out the examples into a single file app and a cross file app

13.12 Changes in 0.6

- Support for JSON and DictFields
- Speeding up QuerySetSelectField with big querysets

13.13 Changes in 0.5

- Added support for all connection settings
- Fixed extended DynamicDocument

13.14 Changes in 0.4

- Added CSRF support and validate_on_save via flask.ext.WTF
- Fixed DateTimeField not required

13.15 Changes in 0.3

- Reverted mongopanel - got knocked out by a merge
- Updated imports paths

13.16 Changes in 0.2

- Added support for password StringField
- Added ModelSelectMultiple

13.17 Changes in 0.1

- Released to PyPi

CHAPTER

FOURTEEN

BSD 3-CLAUSE LICENSE

PYTHON MODULE INDEX

f

INDEX

Symbols

E

F

G

U

W