motorturbine Documentation

Björn Friedrichs

Jul 14, 2018

Contents

Motorturbine is an adapted version of the Motorengine ORM. The main goals are proper asyncio integration as well as a way to have more control over safe updates. Many ORMs suffer from parallelism issues and one big part of this package is to introduce transactions with retry capabilities when updating the fields of a document.

CHAPTER 1

Tutorial

Using Motorturbine is meant to be a streamlined experience, creating an environment where it's the only needed connection to interact with any object in a database.

1.1 Connecting to the database

At first we need to establish a connection to the database. Using *Connection*'s *connect()* all future operations will be made by utilising that connection.

```
Connection.connect(host='localhost', port=27017)
```

1.2 Creating a document

The next step after a global connection is established is to model your documents using the *BaseDocument* class. Modeling is achieved by populating the documents attributes using the supplied *Fields*

```
from motorturbine import BaseDocument, fields

def class Person(BaseDocument):
    name = fields.StringField(default='Nobody')
    age = fields.IntField(required=True)
```

1.3 Working with documents

From here on out each document object can be considered like a typed object.

```
person1 = Person(name="Steve", age=25)

person2 = Person(age=44)
person2.age = 60
```

When all transformations are done objects can be inserted into the database by calling `save()`.

Note: `save()` is a coroutine function and therefore requires awaiting.

```
async def save_person(person):
    await person.save()
```

1.4 Querying objects

The created collections (or document classes) can be queried by using one of the classmethodds `get_object()` or `get_objects()`. These methods will search the collection that is automatically created when inserting a new document. To specify the parameters it is possible to use one or multiple instances of `QueryOperator`.

```
async def get_sixty_plus():
    oldies = await Person.get_objects(age=Gte(60))
    return oldies
```

In this example `motorturbine.queryset.Gte` is used to look for all entries with *Person.age* >= 60.

1.5 Updating fields

Once everything is set up, instead of just setting values directly there is a fancier way to update your fields by utilising mongos inbuilt atomic update capabilities.

Values that are updated this way don't need to match their old state since they just add to the state instead of completely changing it.

```
async def happy_birthday(person):
    person.age = Inc(1)
    await person.save()
```

In this example the `motorturbine.updateset.Inc` operator is used to increase the persons age by one year. For more information about updating see `UpdateOperator`.

CHAPTER 2

Reference

2.1 Documents

Create new documents by subclassing the base class.

2.1.1 BaseDocument

class BaseDocument(***kwargs*)

The BaseDocument is used to create new Documents which can be used to model your data structures.

Simple example using a *StringField* and an *IntField*:

```
class ExampleDocument(BaseDocument):
    name = StringField(default='myname')
    number = IntField(default=0)
```

When instantiating a Document object it is possible to use keyword arguments to initialise its fields to the given values.

```
>>> doc = ExampleDocument(name='Changed My Name', number=15)
>>> print(doc)
<ExampleDocument name='Changed My Name' number=15>
>>> await doc.save()
>>> print(doc)
<ExampleDocument id=ObjectId('$oid') name='Changed My Name' number=15>
```

Raises ***FieldNotFound*** – On access of a non-existent field

Caution: The *id* field is reserved and will be set after a successful save. The field has the same properties as when using an *ObjectIdField*. Will raise an exception if the field is set regardless.

classmethod await get_object(***kwargs*)
A find_one wrapper for `get_objects()`. Queries the collection for a single document. Will return None if there is no or more than one document.

classmethod await get_objects(***kwargs*)
Queries the collection for multiple objects as defined by the supplied filters. For querying Motorturbine supplies its own functionality in form of `QueryOperator`.

await get_reference(*field_name*, *collections=None*)
When using `ReferenceField` this method allows loading the reference by the fields name. Returns *None* if the given field exists but is not a `ReferenceField` type.

Parameters

- **field_name** (*str*) – The name of the ReferenceField
- **collections** (*list*) – optional (*None*) – A list of `BaseDocument` classes. In case you allowed subclassing in a `ReferenceField` you can specify the additional document collections that will be searched if they are not the same as the specified documents type.

Raises **FieldNotFound** – On access of a non-existent field

await save(*limit=0*)
Calling the save method will start a synchronisation process with the database. Every change that was made since the last synchronisation is considered specifically to only update based on the condition that no fields that changed were updated in the meantime. In case that any conflicting fields did update we make sure to pull these changes first and only then update them to avoid critical write errors.

If a document has not been saved before the 'id' field will be set automatically after the update is done.

Parameters **limit** (*int*) – optional *(0)* – The maximum amount of tries before a save operation fails. Can be used as a way to catch problematic state or to probe if the current document has changed yet if set to 1.

Raises **RetryLimitReached** – Raised if limit is reached

to_json()
Returns the entire document as a json dictionary.

2.2 Fields

Used to populate your own Documents. Allow to set defaults, unique indexes and other field specific parameters.

2.2.1 BaseField

class BaseField(***, *default=None*, *required=False*, *unique=False*)
The base class for any field. Used for connecting to the parent document and calling general methods for setting and validating values.

Parameters

- **default** – optional *(None)* – Defines a default value based on the field type.
- **required** (*bool*) – optional *(False)* – Defines if the fields value can be None.
- **unique** (*bool*) – optional *(False)* – Defines if the fields value has to be unique.

Raises **TypeMismatch** – Trying to set a value with the wrong type

2.2.2 FloatField

class FloatField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows a *float* type to be set as its value.

2.2.3 IntField

class IntField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows an *int* type to be set as its value.

2.2.4 StringField

class StringField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows a *str* type to be set as its value.

2.2.5 BooleanField

class BooleanField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows an *bool* type to be set as its value.

2.2.6 ObjectIdField

class ObjectIdField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows a `bson.ObjectId` to be set as its value.

2.2.7 DateTimeField

class DateTimeField(***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field allows multiple types to be set as its value but will always parse them to a `datetime` object.

Accepted types:

- str - Any accepted by `dateutil.parser.parse()` (docs)
- int - Unix timestamp
- float - Unix timestamp
- `datetime.date`
- `datetime.datetime`

Note: Make sure to always use UTC times when trying to insert times to avoid issues between timezones! For example use `datetime.utcnow()` instead of `datetime.now()`

2.2.8 ReferenceField

class ReferenceField(*reference_doc*, ***, *allow_subclass=False*, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.objectid_field.ObjectIdField`

This field allows another Document to be set as its value. A ReferenceField does not auto-insert other fields. Therefore make sure to insert them before you try to set them as a reference.

Parameters

- **reference_doc** (`BaseDocument`) – Sets the document type that will be checked when setting the reference.
- **allow_subclass** (*bool*) – optional (*False*) – Controls whether or not it should be possible to set instances of a subclass of the specified document as a reference.

2.2.9 DocumentField

class DocumentField(*embed_doc*, ***, *default=None*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field allows another Document to be set as its value. Any document inserted as an embedded field will be treated like an object inside of its parent. It enables to create more complex document trees than by just using *`MapField`*.

Example usage:

```
class Identifier(BaseDocument):
    serial = fields.StringField()
    stamp = fields.DateTimeField()
    location = fields.StringField()

class Part(BaseDocument):
    name = fields.StringField()
    ident = fields.DocumentField(Identifier)

now = datetime.utcnow()
ident = Identifier(serial='9X1-33D-52A', stamp=now, location='US')
part = Part(name='Xerxes', ident=ident)

await part.save()
```

In this example an Identifier is attached to each Part that is produced. It wouldn't have been easily possibly to create this structure by using a *`MapField`* because the Identifier is built from more than one data types.

Parameters **embed_doc** (`BaseDocument`) – Sets the document type that will be checked when embedding an instance.

Note: DocumentFields are not only stackable with each other, it is also possible to insert them into a *`ListField`* or *`MapField`*.

2.2.10 ListField

class ListField(*sub_field*, ***, *default=[]*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows a *list* type to be set as its value.

If an entire list is set instead of singular values each entry in the new list has to match the subfield that was set when initialising the field.

Parameters **sub_field** (`BaseField`) – Sets the field type that will be used for the entires of the list.

2.2.11 MapField

class MapField(*value_field*, *key_field=StringField()*, ***, *default={}*, *required=False*, *unique=False*)

Bases: `motorturbine.fields.base_field.BaseField`

This field only allows a *dict* type to be set as its value.

If an entire dict is set instead of singular values each key-value pair in the new dict has to match the subfields that were set when initialising the field.

Parameters **value_field** (`BaseField`) – Sets the field type that will be used for the values of the dict.

2.3 Querying

Operators that allow to create field specific, mongo-like queries.

2.3.1 QueryOperator

class QueryOperator(*value*, *requires_sync=True*)

QueryOperators can be used to automatically generate queries that are understood by mongo. Each of the operators can be used as defined in the mongo manual as they're just a direct mapping. See *`BaseDocument`* to use it with querying methods like *`get_objects()`*.

Note: Please note that because of the overlap in keywords all these classes are capitalised!

2.3.2 Eq

class Eq(*value*, *requires_sync=True*)

Checks for any value that is equal to the given value. Not using it is the default case and functionally the same as just leaving out a QueryOperator completely.

Example usage:

```
>>> await Document.get_objects(num=5)
>>> await Document.get_objects(num=Eq(5))
```

Query:

```
>>> Eq(5)()
{'$eq': 5}
```

2.3.3 Ne

class Ne(*value*, *requires_sync=True*)

Checks for any value that is not equal to the given value.

Example usage:

```
>>> await Document.get_objects(num=Ne(5))
```

Query:

```
>>> Ne(5)()
{'$ne': 5}
```

2.3.4 Lt

class Lt(*value*, *requires_sync=True*)

Checks for any value that is lesser than the given value.

Example usage:

```
>>> await Document.get_objects(num=Lt(5))
```

Query:

```
>>> Lt(5)()
{'$lt': 5}
```

2.3.5 Lte

class Lte(*value*, *requires_sync=True*)

Checks for any value that is lesser than or equal to the given value.

Example usage:

```
>>> await Document.get_objects(num=Lte(5))
```

Query:

```
>>> Lte(5)()
{'$lte': 5}
```

2.3.6 Gt

class Gt(*value*, *requires_sync=True*)

Checks for any value that is greater than the given value.

Example usage:

```
>>> await Document.get_objects(num=Gt(5))
```

Query:

```
>>> Gt(5)()
{'$gt': 5}
```

2.3.7 Gte

class Gte(*value*, *requires_sync=True*)

Checks for any value that is greater than or equal to the given value.

Example usage:

```
>>> await Document.get_objects(num=Gte(5))
```

Query:

```
>>> Gte(5)()
{'$gte': 5}
```

2.3.8 In

class In(*value*, *requires_sync=True*)

Checks for any value that is included in the given value.

Example usage:

```
>>> await Document.get_objects(num=In([1, 4, 5]))
```

Query:

```
>>> In([1, 4, 5])()
{'$in': [1, 4, 5]}
```

2.3.9 Nin

class Nin(*value*, *requires_sync=True*)

Checks for any value that is not included in the given value.

Example usage:

```
>>> await Document.get_objects(num=Nin([1, 4, 5]))
```

Query:

```
>>> Nin([1, 4, 5])()
{'$nin': [1, 4, 5]}
```

2.4 Updating

The Updateset enables updating of fields by using atomic operators.

Note: Makes use of write_bulk to enable the usage of multiple update operators to compress all changes to just on save call on the user side.

2.4.1 UpdateOperator

class UpdateOperator(*update*)

UpdateOperators can be used to automatically generate update queries that are understood by mongo. Each of the operators can be used as defined in the mongo manual as they're just a direct mapping.

Note: Please note that because of the overlap in keywords all these classes are capitalised!

Note: Makes use of write_bulk to enable the usage of multiple update operators to compress all changes to just on save call on the user side.

2.4.2 Set

class Set(*update*)

Is used to set the specified field to any given value. Not using it is the default case and functionally the same as just leaving out an UpdateOperator completely.

Example usage:

```
>>> doc.num = 5
>>> doc.num = Set(5)
```

Query:

```
>>> Set(5)()
{'$set': 5}
```

2.4.3 Inc

Note: Like in mongo Inc can be used with positive and negative numbers. For continuity Dec can also be used and is used for implicit substraction.

class Inc(*update*)

Is used to modify a numeric value by a given amount.

Example usage:

```
>>> doc.num = Inc(5)
>>> doc.num = Inc(-5)
```

Query:

```
>>> Inc(5)()
{'$inc': 5}
```

class Dec(*update*)
Is used to decrease a numeric value.

Example usage:

```
>>> doc.num = Dec(5)
```

Query:

```
>>> Dec(5)()
{'$inc': -5}
```

2.4.4 Max

class Max(*update*)
Update the field to the maximum of database and current value.

Example usage:

```
>>> doc.num = Max(5)
```

Query:

```
>>> Max(5)()
{'$max': 5}
```

2.4.5 Min

class Min(*update*)
Update the field to the minimum of database and current value.

Example usage:

```
>>> doc.num = Min(5)
```

Query:

```
>>> Min(5)()
{'$min': 5}
```

2.4.6 Mul

class Mul(*update*)
Is used to multipy a numeric value by a given amount.

Example usage:

```
>>> doc.num = Mul(5)
```

Query:

```
>>> Mul(5)()
{'$mul': 5}
```

2.4.7 Push

class Push(*update*)

Is used to append a value to a list.

Example usage:

```
>>> doc.num_list = Push(5)
```

Query:

```
>>> Push(5)()
{'$push': 5}
```

2.4.8 Pull

class Pull(*update*)

Is used to pull all entries that match the given value.

Example usage:

```
>>> doc.num_list = Pull(5)
```

Query:

```
>>> Pull(5)()
{'$pull': 5}
```

2.4.9 PullAll

class PullAll(*update*)

Is used to pull all entries that match a value from a list.

Example usage:

```
>>> doc.num_list = PullAll([5, 6, 7])
```

Query:

```
>>> PullAll([5, 6, 7])()
{'$pullAll': [5, 6, 7]}
```

2.5 Connection

A singleton to enable a global connection that can be used by the documents.

2.5.1 Connection

class Connection

This singleton is used to connect motor to your database. When initialising your application call `Connection.connect()` and all subsequent operations on the database will be automatically done by the documents.

classmethod connect(*host='localhost'*, *port=27017*, *database='motorturbine'*)

Connects motorturbine to your database

Parameters

- **host** (`str`) – optional (*'localhost'*) –
- **port** (`int`) – optional (*27017*) –
- **database** (`str`) – optional (*'motorturbine'*) –

2.6 Errors

2.6.1 FieldExpected

class FieldExpected(*received*)

Is raised when a Document is created with an attribute that is not a `BaseField`.

```
>>> raise FieldExpected(str)
Expected instance of BaseField, got str!
```

Parameters received – The received type

2.6.2 TypeMismatch

class TypeMismatch(*expected*, *received*)

Is raised when an incorrect type was supplied.

```
>>> raise TypeMismatch(int, str)
Expected instance of int, got str!
```

Parameters

- **expected** – The expected type.
- **received** – The received type

2.6.3 FieldNotFound

class FieldNotFound(*field_name*, *document*)

Is raised when trying to access a property that isn't present as a field.

```
>>> raise FieldNotFound(doc, 'attr')
Field 'attr' was not found on object     <ExampleDocument name='Changed My Name'␣
↪number=15>.
```

Parameters

- **field_name** (*str*) – Name of the field
- **document** (BaseDocument) – The document that was being accessed.

2.6.4 RetryLimitReached

class RetryLimitReached(*limit*, *document*)

Is raised during the synchronisation process if the specified retry limit is reached.

```
>>> raise RetryLimitReached(10, doc)
Reached the retry limit (10) while trying to save     <ExampleDocument name=
↪'Changed My Name' number=15>.
```

Parameters

- **limit** (*int*) – The retry limit
- **received** (BaseDocument) – The document that couldn't be synced

2.6.5 UnresolvableReference

class UnresolvableReference

Is raised when a reference can not be set/loaded.

```
>>> raise UnresolvableReference()
Only inserted documents can be referenced.
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index