aiomotorengine Documentation

Release 0.9.0.2

ilex

Aug 08, 2017

Contents

AIOMotorEngine is a port (for asyncio's event loop) of the MotorEngine which is port of the MongoEngine. The most part of the code is based on MotorEngine. A few classes and functions were rewritten using async and await for asynchronous code (this syntax was introduced in python 3.5). All tests are rewritten too and seem to pass. AIOMotorEngine does not require Tornado but Motor still does.

CHAPTER 1

Supported Versions

AIOMotorEngine is compatible and tested against python 3.5.

AIOMotorEngine requires MongoDB 2.2+ due to usage of the Aggregation Pipeline.

The tests of compatibility are always run against the current stable version of MongoEngine.

CHAPTER 2

Defining Documents

Defining a new document is as easy as:

```
from aiomotorengine import Document, StringField

class User(Document):
    __collection__ = "users"  # optional. if no collection is specified, class name
↪is used.

    first_name = StringField(required=True)
    last_name = StringField(required=True)

    @property
    def full_name(self):
        return "%s, %s" % (self.last_name, self.first_name)
```

AIOMotorEngine comes baked in with the same fields as MotorEngine.

CHAPTER 3

Example

Let's see how to use AIOMotorEngine:

```
import asyncio
import pymongo
from aiomotorengine import connect, Document, StringField, IntField


class User(Document):
    __collection__ = 'users'

    name = StringField(required=True, unique=True)
    age = IntField()

    def __repr__(self):
        return '<User: {0}({1})>'.format(self.name, self.age)


async def create():
    user = await User.objects.create(name='Python', age=13)
    print(user)
    user2 = User(name='Linux', age=20)
    await user2.save()
    print("_id of user2 is {0}".format(user2._id))

async def query():
    cursor = User.objects.filter(age__gt=10)
    cursor.order_by(User.name, pymongo.DESCENDING)
    cursor.limit(2)
    users = await cursor.find_all()
    print(users)

loop = asyncio.get_event_loop()
connect('xxx', io_loop=loop)
loop.run_until_complete(create())
loop.run_until_complete(query())
```

CHAPTER 4

Contents

Getting Started

Installing

AIOMotorEngine can be installed with pip, using:

```
$ pip install https://github.com/ilex/aiomotorengine/archive/master.zip
```

Connecting to a Database

aiomotorengine.connection.**connect**(*db*, *alias='default'*, ***kwargs*)

Connect to the database specified by the 'db' argument.

Connection settings may be provided here as well if the database is not running on the default port on localhost. If authentication is needed, provide username and password arguments as well.

Multiple databases are supported by using aliases. Provide a separate *alias* to connect to a different instance of **mongod**.

Extra keyword-arguments are passed to Motor when connecting to the database.

```
# create an asyncio event loop

io_loop = asyncio.get_event_loop()
connect("test", host="localhost", port=27017, io_loop=io_loop)  # you only need to␣
↪keep track of the
                                                                # DB instance if you␣
↪connect to multiple databases.
```

Modeling a Document

```
class User(Document):
    first_name = StringField(required=True)
    last_name = StringField(required=True)

class Employee(User):
    employee_id = IntField(required=True)
```

Creating a new instance

```
async def create_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1532)
    emp = await emp.save()
    assert emp is not None
    assert emp.employee_id == 1532

io_loop.run_until_complete(create_employee())
```

Updating an instance

Updating an instance is as easy as changing a property and calling save again:

```
async def update_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1532)
    await emp.save()
    emp.employee_id = 1534
    await emp.save()
    assert emp.employee_id == 1534

io_loop.run_until_complete(update_employee())
```

Getting an instance

To get an object by id, you must specify the ObjectId that the instance got created with. This method takes a string as well and transforms it into a `bson.objectid.ObjectId`.

```
async def load_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1538)
    await emp.save()
    emp2 = await Employee.objects.get(emp._id)
    assert emp2 is not None
    assert emp2.employee_id == 1538

io_loop.run_until_complete(load_employee())
```

Querying collections

To query a collection in mongo, we use the *find_all* method.

If you want to filter a collection, just chain calls to *filter*:

To limit a queryset to just return a maximum number of documents, use the *limit* method:

Ordering the results is achieved with the *order_by* method:

All of these options can be combined to really tune how to get items:

```
async def create_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1538)
    await emp.save()
    # return the first 10 employees ordered by last_name that joined after 2010
    employees = await Employee.objects \
          .limit(10) \
          .order_by("last_name") \
          .filter(last_name="Heynemann") \
          .find_all()

    assert len(employees) > 0
    assert employees[0].last_name == "Heynemann"

io_loop.run_until_complete(create_employee())
```

Counting documents in collections

```
async def count_employees():
    number_of_employees = await Employee.objects.count()
    assert number_of_employees == 0

io_loop.run_until_complete(count_employees())
```

Connecting

Simple Connection

AIOMotorEngine supports connecting to the database using a myriad of options via the *connect* method.

`aiomotorengine.connection.`**`connect`**(*db*, *host="localhost"*, *port=27017*, *io_loop=io_loop*)

Connect to the database specified by the 'db' argument.

Connection settings may be provided here as well if the database is not running on the default port on localhost. If authentication is needed, provide username and password arguments as well.

Multiple databases are supported by using aliases. Provide a separate *alias* to connect to a different instance of **mongod**.

Extra keyword-arguments are passed to Motor when connecting to the database.

```
from aiomotorengine import connect

# instantiate asyncio event loop

io_loop = asyncio.get_event_loop()

# you only need to keep track of the DB instance if you connect to multiple databases.
connect("connecting-test", host="localhost", port=27017, io_loop=io_loop)
```

Replica Sets

aiomotorengine.connection.**connect**(*db*, *host="localhost:27017, localhost:27018"*, *replicaSet="myRs"*, *io_loop=self.io_loop*)

Connect to the database specified by the 'db' argument.

Connection settings may be provided here as well if the database is not running on the default port on localhost. If authentication is needed, provide username and password arguments as well.

Multiple databases are supported by using aliases. Provide a separate *alias* to connect to a different instance of **mongod**.

Extra keyword-arguments are passed to Motor when connecting to the database.

```
from aiomotorengine import connect

# get asyncio event loop

io_loop = asyncio.get_event_loop()
connect("connecting-test", host="localhost:27017,localhost:27018", replicaSet="myRs",␣
↪io_loop=io_loop)
```

The major difference here is that instead of passing a single *host*, you need to pass all the *host:port* entries, comma-separated in the *host* parameter.

You also need to specify the name of the Replica Set in the *replicaSet* parameter (the naming is not pythonic to conform to Motor and thus to pyMongo).

Multiple Databases

aiomotorengine.connection.**connect**(*db*, *alias="db1"*, *host="localhost"*, *port=27017*, *io_loop=io_loop*)

Connect to the database specified by the 'db' argument.

Connection settings may be provided here as well if the database is not running on the default port on localhost. If authentication is needed, provide username and password arguments as well.

Multiple databases are supported by using aliases. Provide a separate *alias* to connect to a different instance of **mongod**.

Extra keyword-arguments are passed to Motor when connecting to the database.

Connecting to multiple databases is as simple as specifying a different alias to each connection.

Let's say you need to connect to an users and a posts databases:

```
from aiomotorengine import connect

# get asyncio event loop

io_loop = asyncio.get_event_loop()

connect("posts", host="localhost", port=27017, io_loop=io_loop)                    # the␣
↪posts database is the default
connect("users", alias="users", host="localhost", port=27017, io_loop=io_loop)   # the␣
↪users database uses an alias

# now when querying for users we'll just specify the alias we want to use
async def go():
    await User.objects.find_all(alias="users")
```

Modeling

AIOMotorEngine uses the concept of models to interact with MongoDB. To create a model we inherit from the *Document* class:

Let's say we need an article model with title, description and published_date:

```
from aiomotorengine.document import Document
from aiomotorengine.fields import StringField, DateTimeField

class Article(Document):
    title = StringField(required=True)
    description = StringField(required=True)
    published_date = DateTimeField(auto_now_on_insert=True)
```

That allows us to create, update, query and remove articles with extreme ease:

```
new_title = "Better Title %s" % uuid4()

async def crud_article():
    article = await Article.objects.create(
        title="Some Article",
        description="This is an article that really matters."
    )

    article.title = new_title
    await article.save()

    articles = await Article.objects.filter(title=new_title).find_all()

    assert len(articles) == 1
    assert articles[0].title == new_title

    number_of_deleted_items = await articles[0].delete()

    assert number_of_deleted_items == 1

io_loop.run_until_complete(crud_article())
```

Base Field

class aiomotorengine.fields.base_field.**BaseField**(*db_field=None*, *default=None*, *required=False*, *on_save=None*, *unique=None*)

This class is the base to all fields. This is not supposed to be used directly in documents.

Available arguments:

- *db_field* - The name this field will have when sent to MongoDB
- *default* - The default value (or callable) that will be used when first creating an instance that has no value set for the field
- *required* - Indicates that if the field value evaluates to empty (using the *is_empty* method) a validation error is raised
- *on_save* - A function of the form *lambda doc, creating* that is called right before sending the document to the DB.

- *unique* - Indicates whether an unique index should be created for this field.

To create a new field, four methods can be overwritten:

- *is_empty* - Indicates that the field is empty (the default is comparing the value to None);
- *validate* - Returns if the specified value for the field is valid;
- *to_son* - Converts the value to the BSON representation required by motor;
- *from_son* - Parses the value from the BSON representation returned from motor.

Available Fields

class aiomotorengine.fields.string_field.**StringField**(*max_length=None*, **args*, ***kw*)

Field responsible for storing text.

Usage:

```
name = StringField(required=True, max_length=255)
```

Available arguments (apart from those in *BaseField*):

- *max_length* - Raises a validation error if the string being stored exceeds the number of characters specified by this parameter

class aiomotorengine.fields.datetime_field.**DateTimeField**(*auto_now_on_insert=False*, *auto_now_on_update=False*, **args*, ***kw*)

Field responsible for storing dates.

Usage:

```
date = DateTimeField(required=True, auto_now_on_insert=True, auto_now_on_
↪update=True)
```

Available arguments (apart from those in BaseField):

- *auto_now_on_insert* - When an instance is created sets the field to datetime.now()
- *auto_now_on_update* - Whenever the instance is saved the field value gets updated to datetime.now()

class aiomotorengine.fields.uuid_field.**UUIDField**(*db_field=None*, *default=None*, *required=False*, *on_save=None*, *unique=None*)

Field responsible for storing `uuid.UUID`.

Usage:

```
name = UUIDField(required=True)
```

class aiomotorengine.fields.boolean_field.**BooleanField**(**args*, ***kw*)

Field responsible for storing boolean values (`bool()`).

Usage:

```
isActive = BooleanField(required=True)
```

BooleanField has no additional arguments available (apart from those in *BaseField*).

Multiple Value Fields

class aiomotorengine.fields.list_field.**ListField**(*base_field=None*, **args*, ***kw*)
Field responsible for storing list.

Usage:

```
posts = ListField(StringField())
```

Available arguments (apart from those in *BaseField*):

•*base_field* - ListField must be another field that describe the items in this list

Embedding vs Referencing

Embedding is very useful to improve the retrieval of data from MongoDB. When you have sub-documents that will always be used when retrieving a document (i.e.: comments in a post), it's useful to have them be embedded in the parent document.

On the other hand, if you need a connection to the current document that won't be used in the main use cases for that document, it's a good practice to use a Reference Field. MotorEngine will only load the referenced field if you explicitly ask it to, or if you set *__lazy__* to *False*.

GeoJson Fields

Saving instances

Creating new instances of a document

The easiest way of creating a new instance of a document is using *Document.objects.create*. Alternatively, you can create a new instance and then call *save* on it.

Updating instances

To update an instance, just make the needed changes to an instance and then call *save*.

Deleting instances

Deleting an instance can be easily accomplished by just calling *delete* on it:

Sometimes, though, the requirements are to remove a few documents (or all of them) at a time. MotorEngine also supports deleting using filters in the document queryset.

Bulk inserting instances

AIOMotorEngine supports bulk insertion of documents by calling the *bulk_insert* method of a queryset with an array of documents:

```
async def create_users():
    users = [
        User(name="Bernardo"),
        User(name="Heynemann")
    ]
    users = await User.objects.bulk_insert(users)
    assert len(users) == 2
    assert users[0]._id
    assert users[1]._id

io_loop.run_until_complete(create_users())
```

Querying

AIOMotorEngine supports a vast array of query operators in MongoDB. There are two ways of querying documents: using the **queryset methods** (filter, filter_not and the likes) or a **Q** object.

Querying with filter methods

Querying with Q

The **Q** object can be combined using python's binary operators | and **&**. Do not confuse those with the **and** and **or** keywords. Those keywords won't call the **__and__** and **__or__** methods in the Q class that are required for the combination of queries.

Let's look at an example of querying for a more specific document. Say we want to find the user that either has a **null** date of last update or is active with a date of last_update lesser than 2010:

```
query = Q(last_update__is_null=True) | (Q(is_active=True) & Q(last_update__
↪lt=datetime(2010, 1, 1, 0, 0, 0)))

query_result = query.to_query(User)

# the resulting query should be similar to:
# {'$or': [{'last_update': None}, {'is_active': True, 'last_update': {'$lt':␣
↪datetime.datetime(2010, 1, 1, 0, 0)}}]}

assert '$or' in query_result

or_query = query_result['$or']
assert len(or_query) == 2
assert 'last_update' in or_query[0]
assert 'is_active' in or_query[1]
assert 'last_update' in or_query[1]
```

Query Operators

Query operators can be used when specified after a given field, like:

```
Q(field_name__operator_name=operator_value)
```

AIOMotorEngine supports the following query operators:

class aiomotorengine.query.exists.**ExistsQueryOperator**
Query operator used to return all documents that have the specified field.

An important reminder is that exists **DOES** match documents that have the specified field **even** if that field value is **NULL**.

For more information on *$exists* go to http://docs.mongodb.org/manual/reference/operator/query/exists/.

Usage:

```
class User(Document):
    name = StringField()

query = Q(name__exists=True)

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'name': {'$exists': True}}
```

class aiomotorengine.query.greater_than.**GreaterThanQueryOperator**
Query operator used to return all documents that have the specified field with a value greater than the specified value.

For more information on *$gt* go to http://docs.mongodb.org/manual/reference/operator/query/gt/.

Usage:

```
class User(Document):
    age = IntField()

query = Q(age__gt=20)

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'age': {'$gt': 20}}
```

class aiomotorengine.query.greater_than_or_equal.**GreaterThanOrEqualQueryOperator**
Query operator used to return all documents that have the specified field with a value greater than or equal to the specified value.

For more information on *$gte* go to http://docs.mongodb.org/manual/reference/operator/query/gte/.

Usage:

```
class User(Document):
    age = IntField()

query = Q(age__gte=21)

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'age': {'$gte': 21}}
```

class aiomotorengine.query.lesser_than.**LesserThanQueryOperator**
Query operator used to return all documents that have the specified field with a value lower than the specified value.

For more information on *$lt* go to http://docs.mongodb.org/manual/reference/operator/query/lt/.

Usage:

```
class User(Document):
    age = IntField()

query = Q(age__lt=20)

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'age': {'$lt': 20}}
```

class aiomotorengine.query.lesser_than_or_equal.**LesserThanOrEqualQueryOperator**
Query operator used to return all documents that have the specified field with a value lower than or equal to the specified value.

For more information on *$lte* go to http://docs.mongodb.org/manual/reference/operator/query/lte/.

Usage:

```
class User(Document):
    age = IntField()

query = Q(age__lte=21)

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'age': {'$lte': 21}}
```

class aiomotorengine.query.in_operator.**InQueryOperator**
Query operator used to return all documents that have the specified field with a value that match one of the values in the specified range.

If the specified field is a ListField, then at least one of the items in the field must match at least one of the items in the specified range.

For more information on *$in* go to http://docs.mongodb.org/manual/reference/operator/query/in/.

Usage:

```
class User(Document):
    age = IntField()
```

```
query = Q(age__in=[20, 21, 22, 23, 24])

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'age': {'$in': [20, 21, 22, 23, 24]}}
```

class aiomotorengine.query.is_null.**IsNullQueryOperator**

Query operator used to return all documents that have the specified field with a null value (or not null if set to False).

This operator uses $exists and $ne for the **not null** scenario.

For more information on *$exists* go to http://docs.mongodb.org/manual/reference/operator/query/exists/.

For more information on *$ne* go to http://docs.mongodb.org/manual/reference/operator/query/ne/.

Usage:

```
class User(Document):
    email = StringField()

query = Q(email__is_null=False)

query_result = query.to_query(User)

# query results should be like:
# {'email': {'$ne': None, '$exists': True}}

assert 'email' in query_result
assert '$ne' in query_result['email']
assert '$exists' in query_result['email']
```

class aiomotorengine.query.not_equal.**NotEqualQueryOperator**

Query operator used to return all documents that have the specified field with a value that's not equal to the specified value.

For more information on *$ne* go to http://docs.mongodb.org/manual/reference/operator/query/ne/.

Usage:

```
class User(Document):
    email = StringField()

query = Q(email__ne="heynemann@gmail.com")

query_result = query.to_query(User)

print(query_result)
```

The resulting query is:

```
{'email': {'$ne': 'heynemann@gmail.com'}}
```

Querying with Raw Queries

Even though AIOMotorEngine strives to provide an interface for queries that makes naming fields and documents transparent, using mongodb raw queries is still supported, both in the filter method and the Q class.

In order to use raw queries, just pass the same object you would use in mongodb:

```
class Address(Document):
    __collection__ = "QueryingWithRawQueryAddress"
    street = StringField()

class User(Document):
    __collection__ = "QueryingWithRawQueryUser"
    addresses = ListField(EmbeddedDocumentField(Address))
    name = StringField()

async def query_user():
    user = User(name="Bernardo", addresses=[Address(street="Infinite Loop")])
    await user.save()
    users = await User.objects.filter({
        "addresses": {
            "street": "Infinite Loop"
        }
    }).find_all()
    assert users[0].name == "Bernardo", users
    assert users[0].addresses[0].street == "Infinite Loop", users

io_loop.run_until_complete(query_user())
```

Python Module Index

a

Index