motorengine Documentation

Release 1.0.0

Bernardo Heynemann

Feb 19, 2018

Contents

MotorEngine is a port of the amazing MongoEngine. 

CHAPTER 1

Supported Versions

MotorEngine is compatible and tested against python 2.7, 3.3, 3.4 and pypy.

MotorEngine 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

Why use MotorEngine?

If you are using tornado, most certainly you don't want your ioLoop to be blocked while doing I/O to mongoDB.

What that means is that MotorEngine allows your tornado instance to keep responding to requests while mongoDB performs your queries.

That's all fine and good, but why choose MotorEngine?

Mainly because MotorEngine helps you in defining consistent documents and makes querying for them really easy.

CHAPTER 3

Defining Documents

Defining a new document is as easy as:

```
from motorengine 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)
```

MotorEngine comes baked in with the same fields as MongoEngine.

CHAPTER 4

What about compatibility?

MotorEngine strives to be 100% compatible with MongoEngine as far as the data in mongoDB goes.

What that means is that it does not feature the same syntax as MongoEngine and vice-versa.

It does not make sense to support the exact same syntax, considering the asynchronous nature of MotorEngine.

Let's see how you query documents using MotorEngine:

```
def get_active_users(callback):
    User.objects.filter(active=True).find_all(callback)

def handle_get_active_users(users):
    # do something with the users
    pass

get_active_users(handle_get_active_users)
```

Or if you are using the new style of doing asynchronous operations in Tornado:

```
def get_active_users():
    users = yield User.objects.filter(active=True).find_all()
    return users
```

CHAPTER 5

Contents

5.1 Getting Started

5.1.1 Installing

MotorEngine can be easily installed with pip, using:

```
$ pip install motorengine
```

If you wish to install it from the source code, you can install it using:

```
$ pip install https://github.com/heynemann/motorengine/archive/master.zip
```

5.1.2 Connecting to a Database

motorengine.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.

```
# instantiate tornado server and apps so we get io_loop instance

io_loop = tornado.ioloop.IOLoop.instance()
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.
```

5.1.3 Modeling a Document

class motorengine.document.**Document**(*_is_partly_loaded=False*, *_reference_loaded_fields=None*, ***kw*)

Base class for all documents specified in MotorEngine.

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

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

5.1.4 Creating a new instance

BaseDocument.**save**(***kwargs*)

Creates or updates the current instance of this document.

Due to the asynchronous nature of MotorEngine, you are required to handle saving in a callback (or using yield method with tornado.concurrent).

```
def create_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1532)
    emp.save(handle_employee_saved)

def handle_employee_saved(emp):
    try:
        assert emp is not None
        assert emp.employee_id == 1532
    finally:
        io_loop.stop()

io_loop.add_timeout(1, create_employee)
io_loop.start()
```

5.1.5 Updating an instance

BaseDocument.**save**(***kwargs*)

Creates or updates the current instance of this document.

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

```
def update_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1532)
    emp.save(handle_employee_created)

def handle_employee_created(emp):
    emp.employee_id = 1534
    emp.save(handle_employee_updated)

def handle_employee_updated(emp):
    try:
        assert emp.employee_id == 1534
    finally:
        io_loop.stop()
```

(continues on next page)

(continued from previous page)

```
io_loop.add_timeout(1, update_employee)
io_loop.start()
```

5.1.6 Getting an instance

QuerySet.**get**(***kwargs*)

Gets a single item of the current queryset collection using it's id.

In order to query a different database, please specify the *alias* of the database to query.

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`.

```
def create_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1538)
    emp.save(handle_employee_saved)

def handle_employee_saved(emp):
    Employee.objects.get(emp._id, callback=handle_employee_loaded)  # every object in␣
↪MotorEngine has an
                                                                    # _id property␣
↪with its ObjectId.

def handle_employee_loaded(emp):
    try:
        assert emp is not None
        assert emp.employee_id == 1538
    finally:
        io_loop.stop()

io_loop.add_timeout(1, create_employee)
io_loop.start()
```

5.1.7 Querying collections

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

QuerySet.**find_all**(***kwargs*)

Returns a list of items in the current queryset collection that match specified filters (if any).

In order to query a different database, please specify the *alias* of the database to query.

Usage:

```
User.objects.find_all(callback=handle_all_users)

def handle_all_users(result):
    # do something with result
    # result is None if no users found
    pass
```

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

QuerySet.**filter**(**arguments*, ***kwargs*)

Filters a queryset in order to produce a different set of document from subsequent queries.

Usage:

```
User.objects.filter(first_name="Bernardo").filter(last_name="Bernardo").find_
↪all(callback=handle_all)
# or
User.objects.filter(first_name="Bernardo", starting_year__gt=2010).find_
↪all(callback=handle_all)
```

The available filter options are the same as used in MongoEngine.

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

`QuerySet.`**`limit`**(*limit*)
Limits the number of documents to return in subsequent queries.

Usage:

```
User.objects.limit(10).find_all(callback=handle_all)  # even if there are 100s of␣
↪users,
                                                      # only first 10 will be␣
↪returned
```

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

`QuerySet.`**`order_by`**(*field_name*, *direction=1*)
Specified the order to be used when returning documents in subsequent queries.

Usage:

```
from motorengine import DESCENDING  # or ASCENDING

User.objects.order_by('first_name', direction=DESCENDING).find_
↪all(callback=handle_all)
```

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

```
def create_employee():
    emp = Employee(first_name="Bernardo", last_name="Heynemann", employee_id=1538)
    emp.save(handle_employee_saved)

def handle_employee_saved(emp):
  # return the first 10 employees ordered by last_name that joined after 2010
  Employee.objects \
          .limit(10) \
          .order_by("last_name") \
          .filter(last_name="Heynemann") \
          .find_all(callback=handle_employees_loaded)

def handle_employees_loaded(employees):
    try:
        assert len(employees) > 0
        assert employees[0].last_name == "Heynemann"
    finally:
        io_loop.stop()

io_loop.add_timeout(1, create_employee)
io_loop.start()
```

5.1.8 Counting documents in collections

QuerySet.**count**(***kwargs*)

Returns the number of documents in the collection that match the specified filters, if any.

```
def get_employees():
  Employee.objects.count(callback=handle_count)

def handle_count(number_of_employees):
    try:
        assert number_of_employees == 0
    finally:
        io_loop.stop()

io_loop.add_timeout(1, get_employees)
io_loop.start()
```

5.2 Connecting

5.2.1 Simple Connection

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

motorengine.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 motorengine import connect

# instantiate tornado server and apps so we get io_loop instance

io_loop = tornado.ioloop.IOLoop.instance()

# 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)
```

5.2.2 Replica Sets

motorengine.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 motorengine import connect

# instantiate tornado server and apps so we get io_loop instance

io_loop = tornado.ioloop.IOLoop.instance()
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).

5.2.3 Multiple Databases

motorengine.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 motorengine import connect

# instantiate tornado server and apps so we get io_loop instance

io_loop = tornado.ioloop.IOLoop.instance()

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
User.objects.find_all(alias="users")
```

5.3 Modeling

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

class motorengine.document.**Document**(*_is_partly_loaded=False*, *_reference_loaded_fields=None*, ***kw*)

Base class for all documents specified in MotorEngine.

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

```
from motorengine.document import Document
from motorengine.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()

def create_article():
    Article.objects.create(
        title="Some Article",
        description="This is an article that really matters.",
        callback=handle_article_created
    )

def handle_article_created(article):
    article.title = new_title
    article.save(callback=handle_article_updated)

def handle_article_updated(article):
    Article.objects.filter(title=new_title).find_all(callback=handle_articles_loaded)

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

    articles[0].delete(callback=handle_article_deleted)

def handle_article_deleted(number_of_deleted_items):
    try:
        assert number_of_deleted_items == 1
    finally:
        io_loop.stop()

io_loop.add_timeout(1, create_article)
io_loop.start()
```

5.3.1 Base Field

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

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.
- *sparse* - Indicates whether a sparse index should be created for this field. This also will not pass empty values to DB.

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.

5.3.2 Available Fields

class motorengine.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 motorengine.fields.datetime_field.**DateTimeField**(*auto_now_on_insert=False*, *auto_now_on_update=False*, *tz=None*, **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()
- ***tz* - Defines the timezone used for auto_now_on_insert and auto_now_on_update and should be enforced on all** values of this datetime field. To interpret all times as UTC use tz=datetime.timezone.utc (Defaults: to None, which means waht you put in comes out again)

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

Field responsible for storing `uuid.UUID`.

Usage:

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

class motorengine.fields.url_field.**URLField**(*db_field=None*, *default=None*, *required=False*, *on_save=None*, *unique=None*, *sparse=False*)

Field responsible for storing URLs.

Usage:

```
name = URLField(required=True)
```

Available arguments (apart from those in *BaseField*): *None*

Note: MotorEngine does not implement the *verify_exists* parameter as MongoEngine due to async http requiring the current io_loop.

---

class motorengine.fields.email_field.**EmailField**(*db_field=None*, *default=None*, *required=False*, *on_save=None*, *unique=None*, *sparse=False*)

Field responsible for storing e-mail addresses.

Usage:

```
name = EmailField(required=True)
```

Available arguments (apart from those in *BaseField*): *None*

class motorengine.fields.int_field.**IntField**(*min_value=None*, *max_value=None*, **args*, ***kw*)

Field responsible for storing integer values (`int()`).

Usage:

```
name = IntField(required=True, min_value=0, max_value=255)
```

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

- *min_value* - Raises a validation error if the integer being stored is lesser than this value
- *max_value* - Raises a validation error if the integer being stored is greather than this value

class motorengine.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*).

class motorengine.fields.float_field.**FloatField**(*min_value=None*, *max_value=None*, **args*, ***kw*)

Field responsible for storing floating-point values (`float()`).

Usage:

```
name = FloatField(required=True, min_value=0.1, max_value=255.6)
```

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

- *min_value* - Raises a validation error if the float being stored is lesser than this value
- *max_value* - Raises a validation error if the float being stored is greather than this value

class motorengine.fields.decimal_field.**DecimalField**(*min_value=None*, *max_value=None*, *precision=2*, *rounding='ROUND_HALF_UP'*, **args*, ***kw*)

Field responsible for storing fixed-point decimal numbers (`decimal.Decimal`).

Usage:

```
import decimal

name = DecimalField(required=True, min_value=None, max_value=None, precision=2,␣
↪rounding=decimal.ROUND_HALF_UP)
```

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

- *min_value* - Raises a validation error if the decimal being stored is lesser than this value
- *max_value* - Raises a validation error if the decimal being stored is greather than this value
- *precision* - Number of decimal places to store.
- *rounding* - The rounding rule from the python decimal library:
  - decimal.ROUND_CEILING (towards Infinity)
  - decimal.ROUND_DOWN (towards zero)
  - decimal.ROUND_FLOOR (towards -Infinity)
  - decimal.ROUND_HALF_DOWN (to nearest with ties going towards zero)
  - decimal.ROUND_HALF_EVEN (to nearest with ties going to nearest even integer)
  - decimal.ROUND_HALF_UP (to nearest with ties going away from zero)
  - decimal.ROUND_UP (away from zero)
  - decimal.ROUND_05UP (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

Note: Decimal field stores the value as a string in MongoDB to preserve the precision.

---

class motorengine.fields.binary_field.**BinaryField**(*max_bytes=None*, **args*, ***kwargs*)

Field responsible for storing binary values.

Usage:

```
name = BinaryField(required=True)
```

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

- *max_bytes* - The maximum number of bytes that can be stored in this field

class motorengine.fields.json_field.**JsonField**(*db_field=None*, *default=None*, *required=False*, *on_save=None*, *unique=None*, *sparse=False*)

Field responsible for storing json objects.

Usage:

```
name = JsonField(required=True)
```

Available arguments (apart from those in *BaseField*): *None*

Note: If ujson is available, MotorEngine will try to use it. Otherwise it will fallback to the json serializer that comes with python.

5.3.3 Multiple Value Fields

class motorengine.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

5.3.4 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*.

class motorengine.fields.embedded_document_field.**EmbeddedDocumentField**(*embedded_document_type=N*, **args*, ***kw*)

Field responsible for storing an embedded document.

Usage:

```
class Comment(Document):
    text = StringField(required=True)

comment = EmbeddedDocumentField(embedded_document_type=Comment)
```

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

- *embedded_document_type* - The type of document that this field accepts as an embedded document.

class motorengine.fields.reference_field.**ReferenceField**(*reference_document_type=None*, **args*, ***kw*)

Field responsible for creating a reference to another document.

Usage:

```
class User(Document):
    name = StringField(required=True)
    email = EmailField(required=True)
```

(continues on next page)

(continued from previous page)

```
owner = ReferenceField(reference_document_type=User)
```

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

- *reference_document_type* - The type of document that this field accepts as a referenced document.

5.4 Saving instances

5.4.1 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.

`QuerySet.`**`create`**`(**kwargs)`

Creates and saved a new instance of the document.

```
def handle_user_created(user):
    try:
        assert user.name == "Bernardo"
    finally:
        io_loop.stop()

def create_user():
    User.objects.create(name="Bernardo", callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

`Document.`**`save`**`(**kwargs)`

Creates or updates the current instance of this document.

```
def handle_user_created(user):
    try:
        assert user.name == "Bernardo"
    finally:
        io_loop.stop()

def create_user():
    user = User(name="Bernardo")
    user.save(callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

5.4.2 Updating instances

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

`Document.`**`save`**`(**kwargs)`

Creates or updates the current instance of this document.

```
def handle_user_created(user):
    user.name = "Heynemann"
    user.save(callback=handle_user_updated)

def handle_user_updated(user):
    try:
        assert user.name == "Heynemann"
    finally:
        io_loop.stop()

def create_user():
    user = User(name="Bernardo")
    user.save(callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

5.4.3 Deleting instances

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

`Document.delete(**kwargs)`
Deletes the current instance of this Document.

```
def handle_user_created(user):
    user.delete(callback=handle_user_deleted)

def handle_user_deleted(number_of_deleted_items):
    try:
        assert number_of_deleted_items == 1
    finally:
        io_loop.stop()

def create_user():
    user = User(name="Bernardo")
    user.save(callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

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.

`QuerySet.delete(**kwargs)`
Removes all instances of this document that match the specified filters (if any).

```
def handle_user_created(user):
    User.objects.filter(name="Bernardo").delete(callback=handle_users_deleted)

def handle_users_deleted(number_of_deleted_items):
    try:
        assert number_of_deleted_items == 1
    finally:
        io_loop.stop()

def create_user():
```

(continues on next page)

(continued from previous page)

```
    user = User(name="Bernardo")
    user.save(callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

5.4.4 Bulk inserting instances

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

QuerySet.**bulk_insert**(*documents*, *callback=None*, *alias=None*)

Inserts all documents passed to this method in one go.

```
def handle_users_inserted(users):
    try:
        assert len(users) == 2
        assert users[0]._id
        assert users[1]._id
    finally:
        io_loop.stop()

def create_users():
    users = [
        User(name="Bernardo"),
        User(name="Heynemann")
    ]
    User.objects.bulk_insert(users, callback=handle_users_inserted)

io_loop.add_timeout(1, create_users)
io_loop.start()
```

5.5 Querying

MotorEngine 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.

5.5.1 Querying with filter methods

QuerySet.**filter**(**arguments*, ***kwargs*)

Filters a queryset in order to produce a different set of document from subsequent queries.

Usage:

```
User.objects.filter(first_name="Bernardo").filter(last_name="Bernardo").find_
↪all(callback=handle_all)
# or
User.objects.filter(first_name="Bernardo", starting_year__gt=2010).find_
↪all(callback=handle_all)
```

The available filter options are the same as used in MongoEngine.

QuerySet.**filter_not**(**arguments*, ***kwargs*)

Filters a queryset to negate all the filters passed in subsequent queries.

Usage:

```
User.objects.filter_not(first_name="Bernardo").filter_not(last_name="Bernardo").
↪find_all(callback=handle_all)
# or
User.objects.filter_not(first_name="Bernardo", starting_year__gt=2010).find_
↪all(callback=handle_all)
```

The available filter options are the same as used in MongoEngine.

5.5.2 Querying with Q

class motorengine.query_builder.node.**Q**(**arguments*, ***query*)

A simple query object, used in a query tree to build up more complex query structures.

```
def handle_users_found(users):
    try:
        assert users[0].name == "Bernardo"
    finally:
        io_loop.stop()

def handle_user_created(user):
    query = Q(name="Bernardo") | Q(age__gt=30)
    users = User.objects.filter(query).find_all(callback=handle_users_found)

def create_user():
    user = User(name="Bernardo", age=32)
    user.save(callback=handle_user_created)

io_loop.add_timeout(1, create_user)
io_loop.start()
```

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]
```

(continues on next page)

(continued from previous page)

```
assert 'is_active' in or_query[1]
assert 'last_update' in or_query[1]
```

5.5.3 Query Operators

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

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

MotorEngine supports the following query operators:

class motorengine.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 motorengine.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 motorengine.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 motorengine.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 motorengine.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 motorengine.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 motorengine.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 motorengine.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'}}
```

5.5.4 Querying with Raw Queries

Even though motorengine 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:

```
import tornado.ioloop

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

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

def create_user():
    user = User(name="Bernardo", addresses=[Address(street="Infinite Loop")])
    user.save(callback=handle_user_created)

def handle_user_created(user):
    User.objects.filter({
        "addresses": {
            "street": "Infinite Loop"
        }
    }).find_all(callback=handle_find_user)

def handle_find_user(users):
    try:
        assert users[0].name == "Bernardo", users
        assert users[0].addresses[0].street == "Infinite Loop", users
    finally:
        io_loop.stop()

io_loop.add_timeout(1, create_user)
io_loop.start()
```

5.5.5 Retrieving a subset of fields

Sometimes a subset of fields on a Document is required, and for efficiency only these should be retrieved from the database. There are some methods that could be used to specify which fields to retrieve. Note that if fields that are not downloaded are accessed, their default value (or None if no default value is provided) will be given.

Projections for reference fields (and a list of reference fields) can be specified too in the same way as for embedded fields. They are applied immediately if *lazy* is *False* or will be applied later after *.load_reference()* will be called otherwise.

Note: You can use *BlogPost.title* notation instead of string value 'title' only for the first level fields. So *BlogPost.author.name* will not work, use string 'author.name' instead. Also *_id* field should be always specified as string '_id'.

Note: It is not possible to save document with projection specified during retrieving. It will raise exception *motorengine.errors.PartlyLoadedDocumentError* as you would try that.

QuerySet.**only**(**fields*)

Load only a subset of this document's fields.

Usage:

```
BlogPost.objects.only(BlogPost.title, "author.name").find_all(...)
```

Note: *only()* is chainable and will perform a union. So with the following it will fetch both: *title* and *comments*:

```
BlogPost.objects.only("title").only("comments").get(...)
```

Note: *only()* does not exclude *_id* field

---

all_fields() will reset any field filters.

Parameters **fields** – fields to include

QuerySet.**exclude**(**fields*)

Opposite to *.only()*, exclude some document's fields.

Usage:

```
BlogPost.objects.exclude("_id", "comments").get(...)
```

Note: *exclude()* is chainable and will perform a union. So with the following it will exclude both: *title* and *author.name*:

```
BlogPost.objects.exclude(BlogPost.title).exclude("author.name").get(...)
```

Note: if *only()* is called somewhere in chain then *exclude()* calls remove fields from the lists of fields specified by *only()* calls:

```
# this will load all fields
BlogPost.objects.only('title').exclude('title').find_all(...)

# this will load only 'title' field
BlogPost.objects.only('title').exclude('comments').get(...)

# this will load only 'title' field
BlogPost.objects.exclude('comments').only('title', 'comments').get(...)

# there is one exception for _id field,
# which will be excluded even if only() is called,
# actually the following is the only way to exclude _id field
BlogPost.objects.only('title').exclude('_id').find_all(...)
```

---

all_fields() will reset any field filters.

Parameters **fields** – fields to exclude

QuerySet.**all_fields**()

Include all fields.

Reset all previously calls of *.only()* or *.exclude()*.

Usage:

```
# this will load 'comments' too
BlogPost.objects.exclude("comments").all_fields().get(...)
```

QuerySet.**fields**(*_only_called=False*, ***kwargs*)

Manipulate how you load this document's fields.

Used by *.only()* and *.exclude()* to manipulate which fields to retrieve. Fields also allows for a greater level of control for example:

Retrieving a Subrange of Array Elements:

You can use the *$slice* operator to retrieve a subrange of elements in an array. For example to get the first 5 comments:

```
BlogPost.objects.fields(slice__comments=5).get(...)
```

or 5 comments after skipping 10 comments:

```
BlogPost.objects.fields(slice__comments=(10, 5)).get(...)
```

or you can also use negative values, for example skip 10 comment from the end and retrieve 5 comments forward:

```
BlogPost.objects.fields(slice__comments=(-10, 5)).get(...)
```

Besides slice, it is possible to include or exclude fields (but it is strongly recommended to use *.only()* and *.exclude()* methods instead):

```
BlogPost.objects.fields(
    slice__comments=5,
    _id=QueryFieldList.EXCLUDE,
    title=QueryFieldList.ONLY
).get(...)
```

Parameters **kwargs** – A dictionary identifying what to include

Python Module Index

m

Index

N

O

Q

R

S

U