Motor-ODM

Release 0.1.dev24

Kim Wittenburg

Apr 13, 2020

TABLE OF CONTENTS

CHAPTER

ONE

OVERVIEW

Motor-ODM is a modern async Object-Document-Mapper for MongoDB. It is based on Pydantic and Motor. It exclusively works with `asyncio`.

CHAPTER

TWO

USING `OBJECTID`

If you are using Pydantic for more than your ODM (e.g. when using FastAPI and want to use the `bson.ObjectId` class you need to tell Pydantic how to handle this class. You can either do this manually or use the handlers from Motor-ODM. To do so all you need to do is make sure that *`motor_odm.document`* is imported before you define your Pydantic models that use `ObjectId`.

CHAPTER

THREE

API-DOCUMENTATION

Motor-ODM consists of several modules and classes all of which are documented in the full *API reference*. This section highlights some classes in order to give you an overview where to start.

motor_odm.document.Document	This is the base class for all documents defined using Motor-ODM.
motor_odm.query.q	Creates a MongoDB query from the specified arguments.

3.1 Package motor_odm

The *motor_odm* package contains all modules for Motor-ODM.

3.1.1 Submodules

Module motor_odm.document

This module contains the base class for interacting with Motor-ODM: *Document*. The *Document* class is the main entry point to Motor-ODM and provides its main interface.

class motor_odm.document.**Document**(***data: Any*)

Bases: pydantic.main.BaseModel

This is the base class for all documents defined using Motor-ODM.

A *Document* is a pydantic model that can be inserted into a MongoDB collection. This class provides an easy interface for interacting with the database. Each document has an *Document.id* (named _id in MongoDB) by default by which it can be uniquely identified in the database. The name of this field cannot be customized however you can override it if you don't want to use ObjectID values for your IDs.

Parameters **abstract** – Mark subclasses as abstract in order to create an abstract document. An abstract document cannot be instantiated but can be subclassed. This enables you to extract common functionality from multiple documents into a common abstract super-document.

classmethod collection() → motor.core.AgnosticCollection

Returns the collection for this *Document*.

The collection uses the codec_options, read_preference, write_concern and read_concern from the document's `Mongo` class.

async classmethod count_documents(*filter: Mapping = None*, **args: Any*, ***kwargs: Any*) → int

Returns the number of documents in this class's collection.

This method is filterable.

classmethod db() → motor.core.AgnosticDatabase
Returns the database that is currently associated with this document.

If no such database exists this returns the database of the parent document (its superclass). If no *Document* class had its *use()* method called to set a db, an AttributeError is raised.

async delete(**args: Any*, ***kwargs: Any*) → bool
Deletes the document from the database.

This method does not modify the instance in any way. args and kwargs are passed to motor's delete_one method.

Returns True if the document was deleted.

async classmethod delete_many(**objects: GenericDocument*) → int
Deletes all specified objects.

Parameters **objects** – All objects to be deleted.

Returns The number of documents deleted.

classmethod find(*filter: DictStrAny = None*, **args: Any*, ***kwargs: Any*) → AsyncIterator[GenericDocument]
Returns an iterable over a cursor returning documents matching filter.

args and kwargs are passed to motor's find method.

async classmethod find_one(*filter: DictStrAny = None*, **args: Any*, ***kwargs: Any*) → Optional[GenericDocument]
Returns a single document from the collection.

async classmethod find_one_and_delete(*filter: DictStrAny = None*, **args: Any*, ***kwargs: Any*) → Optional[GenericDocument]
Finds a document and deletes it.

This method works exactly like pymongo's find_one_and_delete except that this returns a *Document* instance.

async classmethod find_one_and_replace(*filter: DictStrAny*, *replacement: Union[DictStrAny, GenericDocument]*, *return_document: bool = False*, **args: Any*, ***kwargs: Any*) → Optional[GenericDocument]
Finds a document and replaces it.

This method works exactly like pymongo's find_one_and_replace except that this returns a *Document* instance. Note that if you specify return_document=ReturnDocument.AFTER this method will reload the replacement document.

async classmethod find_one_and_update(*filter: DictStrAny*, *update: DictStrAny*, **args: Any*, ***kwargs: Any*) → Optional[GenericDocument]
Finds a document and updates it.

This method works exactly like pymongo's find_one_and_update except that this returns a *Document* instance.

async classmethod init_indexes(*drop: bool = True*, *session: motor.core.AgnosticClientSession = None*, ***kwargs: Any*) → None
Creates the indexes for this collection of documents.

The indexes are specified in the `indexes` field of the `Mongo` class. By default this method makes sure that after the coroutine completes the collection's indexes equal the specified indexes. If you do not want to drop existing indexes you can specify `drop=True` as keyword argument. Note however that this method will always override existing indexes.

Parameters

- **drop** – If `True` all indexes not specified by this collection will be dropped. Default is `True`.
- **session** – A session to use for any database actions.
- **kwargs** – Any keyword arguments are passed to the DB calls. This may be used to specify timeouts etc.

async insert(**args: Any*, ***kwargs: Any*) → bool
Inserts the object into the database.

The object is inserted as a new object. If the document already exists this method will raise an error. Use *save()* instead if you want to update an existing value.

async classmethod insert_many(**objects: GenericDocument*, ***kwargs: Any*) → None
Inserts multiple documents at once.

It is preferred to use this method over multiple *insert()* calls as the performance can be much better.

mongo(*, *include: Union[AbstractSetIntStr, DictIntStrAny] = None*, *exclude: Union[AbstractSetIntStr, DictIntStrAny] = None*) → DictStrAny
Converts this object into a dictionary suitable to be saved to MongoDB.

async reload(**args: Any*, ***kwargs: Any*) → bool
Reloads a document from the database.

Use this method if a model might have changed in the database and you need to retrieve the current version. You do **not** need to call this after inserting a newly created object into the database.

async save(*upsert: bool = True*, **args: Any*, ***kwargs: Any*) → bool
Saves this instance to the database.

By default this method creates the document in the database if it doesn't exist. If you don't want this behavior you can pass `upsert=False`.

Any `args` and `kwargs` are passed to motor's `replace_one` method.

Parameters **upsert** – Whether to create the document if it doesn't exist.

Returns `True` if the document was inserted/updated, `False` if nothing changed. This may also indicate that the document was not changed.

classmethod use(*db: motor.core.AgnosticDatabase*) → None
Sets the database to be used by this *Document*.

The database will also be used by subclasses of this class unless they *use()* their own database. This method has to be invoked before the ODM class can be used.

id: ObjectId = None
The document's ID in the database.

By default this field is of type `ObjectId` but it can be overridden to supply your own ID types. Note that if you intend to override this field you **must** set its alias to `_id` in order for your IDs to be recognized as such by MongoDB.

class motor_odm.document.**Mongo**
Bases: object

This class defines the defaults for collection configurations.

Each collection (defined by a subclass of *Document*) can override these using an inner class named Mongo. Attributes are implicitly and transitively inherited from the Mongo classes of base classes.

abstract: bool = False
Whether the document is abstract.

The value for this field cannot be specified in the Mongo class but as a keyword argument on class creation.

codec_options: Optional[bson.codec_options.CodecOptions] = None
The codec options to use when accessing the collection.

Defaults to the database's *codec_options*.

collection: Optional[str] = None
The name of the collection for a document. This attribute is required.

indexes: List[pymongo.operations.IndexModel] = []
A list of indexes to create for the collection.

read_concern: Optional[pymongo.read_concern.ReadConcern] = None
The read concern to use when accessing the collection.

Defaults to the database's *read_concern*.

read_preference: Optional[pymongo.read_preferences.ReadPreference] = None
The read preference to use when accessing the collection.

Defaults to the database's *read_preference*.

write_concern: Optional[pymongo.write_concern.WriteConcern] = None
The write concern to use when accessing the collection.

Defaults to the database's *write_concern*.

Module motor_odm.encoders

BSON encoders for common python types.

This module contains a collection of bson.codec_options.TypeEncoder subclasses for common python types such as sets. Note that these encoders are provided as a convenience but are not used automatically. If you want to use sets in your documents you have to provide the appropriate codec_options to the MongoDB client, database, collection or function.

class motor_odm.encoders.**SetEncoder**
Bases: bson.codec_options.TypeEncoder

BSON support for python set.

This encoder encodes a set in form of a list. The list is not converted back into a set automatically but if you are using the *Document* class this is done upon initialization of your model.

python_type
alias of builtins.set

transform_python
alias of builtins.list

class motor_odm.encoders.**FrozensetEncoder**
Bases: bson.codec_options.TypeEncoder

BSON support for python frozenset.

This encoder encodes a `frozenset` in form of a `list`. The list is not converted back into a set automatically but if you are using the *Document* class this is done upon initialization of your model.

python_type
alias of `builtins.frozenset`

transform_python
alias of `builtins.list`

Module `motor_odm.fixtures`

This module contains patches for some frameworks to make Motor-ODM work as one would expect. Expect some more or less ugly hacks here...

Note that all patches are applied automatically at **import time**.

`motor_odm.fixtures.`**`patch_fastapi`**`()` → None
Patches the FastAPI framework to support models based on Pydantic. By default FastAPI routes handle Pydantic models specially. This patch removes the special case for subclasses of *Document*.

Module `motor_odm.helpers`

This module contains various supporting functions that can be used independently of the Motor-ODM framework. Some of these utilities can be found in similar form in other packages or frameworks and are adapted here to reduce the number of dependencies.

`motor_odm.helpers.`**`inherit_class`**`(`*name: str*, *self: Optional[T]*, *parent: T*, *merge: Iterable[str] = None*`)` → T
Performs a pseudo-inheritance by creating a new class that inherits from `self` and `parents`. This is useful to support intuitive inheritance on inner classes (typically named `Meta`).

Note that this method neither returns `self` nor any of the `parents` but a new type that inherits from both.

Parameters

- **`name`** – The name of the newly created type.
- **`self`** – The primary base class (fields in this class take preference over the `parents`' fields.
- **`parent`** – The secondary base class (a pseudo-parent of `self`).
- **`merge`** – A list of fields that should not be replaces during inheritance but merged. This only works for some types.

Returns A new type inheriting from `self` and `parents`.

`motor_odm.helpers.`**`merge_values`**`(`*value1: Any*, *value2: Any*`)` → Any
Merges two values.

This method works only for specific collection types (namely lists, dicts and sets). For other values a `ValueError` is raised.

The type of the resulting value is determined by the type of `value2`, however `value1` may override some of the contents in `value2` (e.g. replace values for dict keys).

`motor_odm.helpers.`**`monkey_patch`**`(`*cls: Union[type, module]*, *name: Optional[str] = None*`)` → Callable[[C], C]
Monkey patches class or module by adding to it decorated function. Anything overwritten can be accessed via a `.original` attribute of the decorated object.

Parameters

- **cls** – The class or module to be patched.
- **name** – The name of the attribute to be patched.

Returns A decorator that monkey patches `cls.name` and returns the decorated function.

Module motor_odm.indexes

class motor_odm.indexes.**IndexManager**(*collection: motor.core.AgnosticCollection*, *session: motor.core.AgnosticClientSession*, ***kwargs: Any*)

Bases: object

An `IndexManager` instance can create a specific state concerning indexes.

Note: The `IndexManager` is used internally by *Document*. Normally there is no need to use this class manually.

---

async ensure_indexes(*indexes: Iterable[pymongo.operations.IndexModel]*, *drop: bool = True*) → None

Ensures that the specified `indexes` exist in the databse.

This method communicates with the database several times to compare the specified `indexes` to the indexes already present in the database. If an index already exists it is not recreated. If an index exists with different options (name, uniqueness, ...) it is dropped and recreated.

Any indexes in the database that are not specified in `indexes` will be dropped unless `drop=False` is specified.

static equal(*index: pymongo.operations.IndexModel*, *db_index: bson.son.SON*) → bool

Compares the specified `index` and `db_index`.

This method return `True` if the `index` specification can be considered equal to the existing `db_index` in the database.

get_db_index(*spec: bson.son.SON*) → Optional[bson.son.SON]

Fetches the database index matching the `spec`.

This method does not communicate with the database. You have to have called *load_db_indexes()* before.

Returns An index from the database or `None` if no index matching `spec` exists.

async load_db_indexes() → None

Loads the existing indexes from the database.

Module motor_odm.query

class motor_odm.query.**Query**(**args: Any*, ***kwargs: Any*)

Bases: dict, typing.Generic

A MongoDB query.

Queries behave like ordinary dictionaries (in fact they inherit from `dict`). However they offer some additional convenience related to MongoDB queries. Most notably they can be constructed conveniently from keyword arguments. See the documentation on *q()* for details.

This class also offers some factory methods for special queries such as using a JSON Schema.

add_expression(*field: str*, *value: Any*, *op: str = None*) → None
Adds a single expression to this query.

An expression is a constraint for a single field. This method modifies the query to add a constraint for the specified `field` to be equal to `value`. If `op` is specified it is used instead of the default `$eq` operator.

Raises **KeyError** – If `field` has already a value for `op` that is not equal to `value`.

comment(*comment: str*) → *motor_odm.query.Query*
Adds a comment to this query.

classmethod expr(*expression: dict*) → *motor_odm.query.Query*
Constructs an `$expr` query.

extend(***kwargs: DictStrAny*) → None
Adds fields to this query.

This method adds the same keys and values that you would get using the *q()* function with only keyword arguments. See the documentation on that method for details.

classmethod schema(*schema: DictStrAny*) → Query
Constructs a `$jsonSchema` query.

classmethod text(*search: str*, *language: str = None*, *case_sensitive: bool = None*, *diacritic_sensitive: bool = None*) → *motor_odm.query.Query*
Constructs a `$text` query using the specified arguments.

classmethod where(*where: Union[str, bson.code.Code]*) → *motor_odm.query.Query*
Constructs a `$where` query.

motor_odm.query.**q**(**args: Any*, ***kwargs: Any*) → *motor_odm.query.Query*
Creates a MongoDB query from the specified arguments.

The query can be used to filter documents in Motor-ODM or even directly in Motor or PyMongo. This function is the preferred way of constructing queries. You can use special keyword arguments to construct more complex queries.

One of the most common cases is a query by ID. Specify the ID as the only positional argument:

```
>>> q(123)
{'_id': 123}
```

If you pass `None` as the single id value, a query will be constructed that matches nothing. >>> q(None) {'X': {'$in': []}}

You can also create a query that matches an ID in a list of IDs by specifying multiple positional arguments, each of which will be treated as a possible ID. In this case `None` values will simply be ignored. >>> q(123, None, "ABC") {'_id': {'$in': [123, 'ABC']}}

Instead of querying the ID of a document you most likely need to filter documents based on their fields. You can do this by providing the respective keyword arguments. You can combine positional and keyword arguments if you need to. In the simples case we want to create a query that filters on the value of one or more fields:

```
>>> q(name="John", age=20)
{'name': 'John', 'age': 20}
```

You can also use MongoDB query operators to create more complex queries:

```
>>> q(age__gt=20, age__lt=100)
{'age': {'$gt': 20, '$lt': 100}}
```

Lastly you can combine queries with `&`, `|` and `^`. The `^` operator means *nor* in this case.

```
>>> (q(age=20) & q(name="John")) | q(age=21)
{'$or': [{'$and': [{'age': 20}, {'name': 'John'}]}, {'age': 21}]}
```

3.2 Index

PYTHON MODULE INDEX

m

INDEX

P

Q

R

S

T

U

W