Motor-ODM

Release 0.1.dev22

Kim Wittenburg

Apr 11, 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.

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.**DocumentMetaclass**
Bases: pydantic.main.ModelMetaclass

The meta class for *Document*. Ensures that the Mongo class is automatically inherited.

class motor_odm.document.**Document**
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.

classmethod all(*db_filter: Query = None*, ***kwargs: Any*) → AsyncIterator[GenericDocument]
Returns multiple documents from the collection.

This method is filterable.

async classmethod batch_insert(**objects: GenericDocument*) → None
Inserts multiple documents at once.

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

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(*db_filter: Query = None*, ***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.

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

classmethod find(*db_filter: Query = None*, ***kwargs: Any*) → AsyncIterator[GenericDocument]
Returns multiple documents from the collection.

This method is filterable.

async classmethod get(*db_filter: Query = None*, ***kwargs: Any*) → Optional[GenericDocument]
Returns a single document from the collection.

This method is filterable.

async insert() → None
Inserts the object into the database.

The object is inserted as a new object.

async reload() → None
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.

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.

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]*, **parents: T*) → 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.
- **parents** – The secondary base classes. Field preferences are determined by the order of the parent classes.

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

`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.query`

This module contains functions for building MongoDB queries.

`motor_odm.query.`**create_query**(*db_filter: Query = None*, ***kwargs: Any*) → DictStrAny

Creates a MongoDB query from the specified arguments. This helper can be invoked in three ways (which can also be combined):

Create a filter from keyword arguments. The arguments are transformed into a `dict` and returned verbatim.

```
>>> create_query(username="john")
{'username': 'john'}
```

For more complex cases you can also supply your own filter as a `dict`.

```
>>> create_query({"username": "john"})
{'username': 'john'}
```

You can also combine both methods.

```
>>> create_query({"username": "john"}, password="abc123")
{'password': 'abc123', 'username': 'john'}
```

One special case that is often required is looking up objects by their ID. You can do this by passing a value for the ID as the first positional argument. This cannot be combined with other advanced filters but can accept additional keyword arguments.

```
>>> create_query("john", password="abc123")
{'password': 'abc123', '_id': 'john'}
```

Parameters

- **db_filter** – Either a dict that is included as a filter or any other type that is used as an `_id` query.

- **kwargs** – Filter arguments. Keyword arguments are preceeded by the `db_filter` parameter.

Returns A `dict` that can be used to filter a MongoDB collection.

3.2 Index

PYTHON MODULE INDEX

m

INDEX