PyMODM Documentation

Release 0.3.0

Luke Lovett

December 05, 2016

Contents

CHAPTER 1

Overview

PyMODM is a generic ODM on top of PyMongo, the MongoDB Python driver. This documentation attempts to explain everything you need to know to use **PyMODM**. In addition to this documentation, you can also check out example code as part of the PyMODM project on Github.

Contents:

Getting Started with PyMODM Start here for a basic overview of using PyMODM.

API Documentation The complete API documentation.

1.1 API Documentation

Welcome to the PyMODM API documentation.

1.1.1 Connecting

Tools for managing connections in MongoModels.

`pymodm.connection.`**`connect`**(*mongodb_uri*, *alias='default'*)

Register a connection to MongoDB, optionally providing a name for it.

Parameters

- *mongodb_uri*: A MongoDB connection string. Any options may be passed within the string that are supported by PyMongo. *mongodb_uri* must specify a database, which will be used by any `MongoModel` that uses this connection.
- *alias*: An optional name for this connection, backed by a `MongoClient` instance that is cached under this name. You can specify what connection a MongoModel uses by specifying the connection's alias via the *connection_alias* attribute inside their *Meta* class. Switching connections is also possible using the `switch_connection` context manager. Note that calling *connect()* multiple times with the same alias will replace any previous connections.

1.1.2 Defining Models

class `pymodm.`**`MongoModel`**(**args*, ***kwargs*)

Base class for all top-level models.

A MongoModel definition typically includes a number of field instances and possibly a `Meta` class attribute that provides metadata or settings specific to the model.

MongoModels can be instantiated either with positional or keyword arguments. Positional arguments are bound to the fields in the order the fields are defined on the model. Keyword argument names are the same as the names of the fields:

```
from pymongo.read_preferences import ReadPreference

class User(MongoModel):
    email = fields.EmailField(primary_key=True)
    name = fields.CharField()

    class Meta:
        # Read from secondaries.
        read_preference = ReadPreference.SECONDARY

# Instantiate User using positional arguments:
jane = User('jane@janesemailaddress.net', 'Jane')
# Keyword arguments:
roy = User(name='Roy', email='roy@roysemailaddress.net')
```

The following metadata attributes are available:

- *connection_alias*: The alias of the connection to use for the moel.
- *collection_name*: The name of the collection to use. By default, this is the same name as the model, converted to snake case.
- *codec_options*: An instance of `CodecOptions` to use for reading and writing documents of this model type.
- *final*: Whether to restrict inheritance on this model. If `True`, the `_cls` field will not be stored in the document. `False` by default.
- *cascade*: If `True`, save all *`MongoModel`* instances this object references when *`save()`* is called on this object.
- *read_preference*: The `ReadPreference` to use when reading documents.
- *read_concern*: The `ReadConcern` to use when reading documents.
- *write_concern*: The `WriteConcern` to use for write operations.
- *indexes*: This is a list of `IndexModel` instances that describe the indexes that should be created for this model. Indexes are created when the class definition is evaluated.

Note: Creating an instance of MongoModel does not create a document in the database.

---

clean()

Run custom validation rules run when *`full_clean()`* is called.

This is an abstract method that can be overridden to validate the *`MongoModel`* instance as a whole. Custom field validation is better done by passing a validator to the *validators* parameter in a field's constructor.

example:

```
class Vacation(MongoModel):
    destination = fields.CharField(choices=('HAWAII', 'DETROIT'))
    travel_method = fields.CharField(
        choices=('PLANE', 'CAR', 'BOAT'))
```

```
        def clean(self):
            # Custom validation that requires looking at several fields.
            if (self.destination == 'HAWAII' and
                    self.travel_method == 'CAR'):
                raise ValidationError('Cannot travel to Hawaii by car.')
```

clean_fields(*exclude=None*)
Validate the values of all fields.

This method will raise a *ValidationError* that describes all issues with each field, if any field fails to pass validation.

Parameters

- *exclude*: A list of fields to exclude from validation.

delete()
Delete this object from MongoDB.

from_document(*document*)
Construct an instance of this class from the given document.

Parameters

- *document*: A Python dictionary describing a MongoDB document. Keys within the document must be named according to each model field's *mongo_name* attribute, rather than the field's Python name.

full_clean(*exclude=None*)
Validate this *MongoModel*.

This method calls *clean_fields()* to validate the values of all fields then *clean()* to apply any custom validation rules to the model as a whole.

Parameters

- *exclude*: A list of fields to exclude from validation.

is_valid()
Return `True` if the data in this Model is valid.

This method runs the *full_clean()* method and returns `True` if no ValidationError was raised.

pk
An alias for the primary key (called *_id* in MongoDB).

refresh_from_db(*fields=None*)
Reload this object from the database, overwriting local field values.

Parameters

- *fields*: An iterable of fields to reload. Defaults to all fields.

> **Warning:** This method will reload the object from the database, possibly with only a subset of fields. Calling *save()* after this may revert or unset fields in the database.

classmethod **register_delete_rule**(*related_model*, *related_field*, *rule*)
Specify what to do when an instance of this class is deleted.

Parameters

- *related_model*: The class that references this class.

- *related_field*: The name of the field in `related_model` that references this class.
- *rule*: The delete rule. See *`ReferenceField`* for details.

save(*cascade=None*, *full_clean=True*, *force_insert=False*)
Save this document into MongoDB.

If there is no value for the primary key on this Model instance, the instance will be inserted into MongoDB. Otherwise, the entire document will be replaced with this version (upserting if necessary).

Parameters

- *cascade*: If `True`, all dereferenced MongoModels contained in this Model instance will also be saved.
- *full_clean*: If `True`, the *`full_clean()`* method will be called before persisting this object.
- *force_insert*: If `True`, always do an insert instead of a replace. In this case, *save* will raise `DuplicateKeyError` if a document already exists with the same primary key.

Returns This object, with the *pk* property filled in if it wasn't already.

to_son()
Get this Model back as a `SON` object.

Returns SON representing this object as a MongoDB document.

class pymodm.**EmbeddedMongoModel**(**args*, ***kwargs*)
Base class for models that represent embedded documents.

clean()
Run custom validation rules run when *`full_clean()`* is called.

This is an abstract method that can be overridden to validate the *`MongoModel`* instance as a whole. Custom field validation is better done by passing a validator to the *validators* parameter in a field's constructor.

example:

```
class Vacation(MongoModel):
    destination = fields.CharField(choices=('HAWAII', 'DETROIT'))
    travel_method = fields.CharField(
        choices=('PLANE', 'CAR', 'BOAT'))

    def clean(self):
        # Custom validation that requires looking at several fields.
        if (self.destination == 'HAWAII' and
                self.travel_method == 'CAR'):
            raise ValidationError('Cannot travel to Hawaii by car.')
```

clean_fields(*exclude=None*)
Validate the values of all fields.

This method will raise a *`ValidationError`* that describes all issues with each field, if any field fails to pass validation.

Parameters

- *exclude*: A list of fields to exclude from validation.

from_document(*document*)
Construct an instance of this class from the given document.

Parameters

- *document*: A Python dictionary describing a MongoDB document. Keys within the document must be named according to each model field's *mongo_name* attribute, rather than the field's Python name.

full_clean(*exclude=None*)
Validate this *MongoModel*.

This method calls *clean_fields()* to validate the values of all fields then *clean()* to apply any custom validation rules to the model as a whole.

Parameters

- *exclude*: A list of fields to exclude from validation.

to_son()
Get this Model back as a SON object.

Returns SON representing this object as a MongoDB document.

1.1.3 Model Fields

class pymodm.base.fields.**MongoBaseField**(*verbose_name=None*, *mongo_name=None*, *primary_key=False*, *blank=False*, *required=False*, *default=None*, *choices=None*, *validators=None*)
Base class for all MongoDB Model Field types.

Create a new Field instance.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *primary_key*: If `True`, this Field will be used for the `_id` field when stored in MongoDB. Note that the *mongo_name* of the primary key field cannot be changed from `_id`.
- *blank*: If `True`, allow this field to have an empty value.
- *required*: If `True`, do not allow this field to be unspecified.
- *default*: The default value to use for this field if no other value has been given.
- *choices*: A list of possible values for the field. This can be a flat list, or a list of 2-tuples consisting of an allowed field value and a human-readable version of that value.
- *validators*: A list of callables used to validate this Field's value.

PyMongo ODM Field Definitions.

class pymodm.fields.**GenericIPAddressField**(*verbose_name=None*, *mongo_name=None*, *protocol=2*, ***kwargs*)
A field that stores IPV4 and/or IPV6 addresses.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *protocol*: What protocol this Field should accept. This should be one of the following:
  - *GenericIPAddressField.IPV4*
  - *GenericIPAddressField.IPV6*

- *GenericIPAddressField.BOTH* (default).

See also:

constructor for *MongoBaseField*

IPV4 = 0
Accept IPv4 addresses only.

IPV6 = 1
Accept IPv6 addresses only.

BOTH = 2
Accept both IPv4 and IPv6 addresses.

class pymodm.fields.**ReferenceField**(*model*, *on_delete=0*, *verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that references another document within a document.

Parameters

- *model*: The class of *MongoModel* that this field references.
- *on_delete*: The action to take (if any) when the referenced object is deleted. The delete rule should be one of the following:
- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
  - *ReferenceField.DO_NOTHING* (default).
  - *ReferenceField.NULLIFY*
  - *ReferenceField.CASCADE*
  - *ReferenceField.DENY*
  - *ReferenceField.PULL*

See also:

constructor for *MongoBaseField*

DO_NOTHING = 0
Don't do anything upon deletion.

NULLIFY = 1
Set the reference to `None` upon deletion.

CASCADE = 2
Delete documents associated with the reference.

DENY = 3
Disallow deleting objects that are still referenced.

PULL = 4
Pull the reference of the deleted object out of a *ListField*

class pymodm.fields.**CharField**(*verbose_name=None*, *mongo_name=None*, *min_length=None*, *max_length=None*, ***kwargs*)
A field that stores unicode strings.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

- *min_length*: The required minimum length of the string.
- *max_length*: The required maximum length of the string.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**IntegerField**(*verbose_name=None*, *mongo_name=None*, *min_value=None*, *max_value=None*, ***kwargs*)

A field that stores a Python int.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *min_value*: The minimum value that can be stored in this field.
- *max_value*: The maximum value that can be stored in this field.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**BigIntegerField**(*verbose_name=None*, *mongo_name=None*, *min_value=None*, *max_value=None*, ***kwargs*)

A field that always stores and retrieves numbers as bson.int64.Int64.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *min_value*: The minimum value that can be stored in this field.
- *max_value*: The maximum value that can be stored in this field.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**ObjectIdField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores ObjectIds.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**BinaryField**(*verbose_name=None*, *mongo_name=None*, *subtype=0*, ***kwargs*)

A field that stores binary data.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *subtype*: A subtype listed in the binary module.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**BooleanField**(*verbose_name=None*, *mongo_name=None*, *primary_key=False*, *blank=False*, *required=False*, *default=None*, *choices=None*, *validators=None*)

A field that stores boolean values.

Create a new Field instance.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *primary_key*: If `True`, this Field will be used for the `_id` field when stored in MongoDB. Note that the *mongo_name* of the primary key field cannot be changed from `_id`.
- *blank*: If `True`, allow this field to have an empty value.
- *required*: If `True`, do not allow this field to be unspecified.
- *default*: The default value to use for this field if no other value has been given.
- *choices*: A list of possible values for the field. This can be a flat list, or a list of 2-tuples consisting of an allowed field value and a human-readable version of that value.
- *validators*: A list of callables used to validate this Field's value.

class pymodm.fields.**DateTimeField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores `datetime` objects.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**Decimal128Field**(*verbose_name=None*, *mongo_name=None*, *min_value=None*, *max_value=None*, ***kwargs*)

A field that stores `Decimal128` objects.

Note: This requires MongoDB >= 3.4.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *min_value*: The minimum value that can be stored in this field.
- *max_value*: The maximum value that can be stored in this field.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**EmailField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores email addresses.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**FileField**(*verbose_name=None*, *mongo_name=None*, *storage=None*, ***kwargs*)
A field that stores files.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *storage*: The *Storage* implementation to use for saving and opening files.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**ImageField**(*verbose_name=None*, *mongo_name=None*, *storage=None*, ***kwargs*)
A field that stores images.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *storage*: The *Storage* implementation to use for saving and opening files.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**FloatField**(*verbose_name=None*, *mongo_name=None*, *min_value=None*, *max_value=None*, ***kwargs*)
A field that stores a Python *float*.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *min_value*: The minimum value that can be stored in this field.
- *max_value*: The maximum value that can be stored in this field.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**URLField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores URLs.

Parameters

- *verbose_name*: A human-readable name for the Field.

- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**UUIDField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores UUID objects.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**RegularExpressionField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores MongoDB regular expression types.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**JavaScriptField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores JavaScript code.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**TimestampField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores a BSON timestamp.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**DictField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores a regular Python dictionary.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**OrderedDictField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores a OrderedDict.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**ListField**(*field=None*, *verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores a list.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.
- *field*: The Field type of all items in this list.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**PointField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores the GeoJSON 'Point' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list that simply describes (longitude, latitude) in that order.

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**LineStringField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores the GeoJSON 'LineString' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of coordinate points (each a list of two coordinates).

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**PolygonField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)
A field that stores the GeoJSON 'Polygon' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of LineStrings (each a list of Points).

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**MultiPointField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores the GeoJSON 'MultiPoint' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of Points (a list containing two coordinates).

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**MultiLineStringField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores the GeoJSON 'MultiLineString' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of LineStrings (each a list of Points).

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**MultiPolygonField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores the GeoJSON 'MultiPolygonField' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of LineStrings (each a list of Points).

Parameters

- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**GeometryCollectionField**(*verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores the GeoJSON 'GeometryCollection' type.

Values may be assigned to this field that are already in GeoJSON format, or you can assign a list of geometries, where each geometry is a GeoJSON document.

Parameters

- *verbose_name*: A human-readable name for the Field.

- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**EmbeddedDocumentField**(*model*, *verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores a document inside another document.

Parameters

- *model*: A *EmbeddedMongoModel*, or the name of one, as a string.
- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

class pymodm.fields.**EmbeddedDocumentListField**(*model*, *verbose_name=None*, *mongo_name=None*, ***kwargs*)

A field that stores a list of documents within a document.

All documents in the list must be of the same type.

Parameters

- *model*: A *EmbeddedMongoModel*, or the name of one, as a string.
- *verbose_name*: A human-readable name for the Field.
- *mongo_name*: The name of this field when stored in MongoDB.

See also:

constructor for *MongoBaseField*

1.1.4 Managers

class pymodm.manager.**BaseManager**

Abstract base class for all Managers.

BaseManager has no underlying *QuerySet* implementation. To extend this class into a concrete class, a *QuerySet* implementation must be provided by calling *from_queryset()*:

```
class MyQuerySet(QuerySet):
    ...

MyManager = BaseManager.from_queryset(MyQuerySet)
```

Extending this class by calling *from_queryset* creates a new Manager class that wraps only the methods from the given *QuerySet* type (and not from the default *QuerySet* implementation).

See also:

The default *Manager*.

contribute_to_class(*cls*, *name*)

creation_order

The order in which this Manager instance was created.

classmethod **from_queryset**(*queryset_class*, *class_name=None*)
Create a Manager that delegates methods to the given *QuerySet* class.

The Manager class returned is a subclass of this Manager class.

Parameters

- *queryset_class*: The QuerySet class to be instantiated by the Manager.
- *class_name*: The name of the Manager class. If one is not provided, the name of the Manager will be *XXXFromYYY*, where *XXX* is the name of this Manager class, and YYY is the name of the QuerySet class.

get_queryset()
Get a QuerySet instance.

class pymodm.manager.**Manager**
The default manager used for *MongoModel* instances.

This implementation of *BaseManager* uses *QuerySet* as its QuerySet class.

This Manager class (accessed via the `objects` attribute on a *MongoModel*) is used by default for all MongoModel classes, unless another Manager instance is supplied as an attribute within the MongoModel definition.

Managers have two primary functions:

1. Construct *QuerySet* instances for use when querying or working with *MongoModel* instances in bulk.
2. Define collection-level functionality that can be reused across different MongoModel types.

If you created a custom QuerySet that makes certain queries easier, for example, you will need to create a custom Manager type that returns this queryset using the *from_queryset()* method:

```
class UserQuerySet(QuerySet):
    def active(self):
        '''Return only active users.'''
        return self.raw({"active": True})

class User(MongoModel):
    active = fields.BooleanField()
    # Add our custom Manager.
    users = Manager.from_queryset(UserQuerySet)
```

In the above example, we added a *users* attribute on *User* so that we can use the *active* method on our new QuerySet type:

```
active_users = User.users.active()
```

If we wanted every method on the QuerySet to examine active users *only*, we can do that by customizing the Manager itself:

```
class UserManager(Manager):
    def get_queryset(self):
        # Override get_queryset, so that every QuerySet created will
        # have this filter applied.
        return super(UserManager, self).get_queryset().raw(
            {"active": True})

class User(MongoModel):
    active = fields.BooleanField()
    users = UserManager()

active_users = User.users.all()
```

1.1.5 QuerySet

class pymodm.queryset.**QuerySet**(*model=None*, *query=None*)
The default QuerySet type.

QuerySets handle queries and allow working with documents in bulk.

Parameters

- *model*: The *MongoModel* class to be produced by the QuerySet.
- *query*: The MongoDB query that filters the QuerySet.

aggregate(**pipeline*, ***kwargs*)
Perform a MongoDB aggregation.

Any query, projection, sort, skip, and limit applied to this QuerySet will become aggregation pipeline stages in that order *before* any additional stages given in *pipeline*.

Parameters

- *pipeline*: Additional aggregation pipeline stages.
- *kwargs*: Keyword arguments to pass down to PyMongo's `aggregate()` method.

Returns A `CommandCursor` over the result set.

example:

```
>>> # Apply a filter before aggregation.
>>> qs = Vacation.objects.raw({'travel_method': 'CAR'})
>>> # Run aggregation pipeline.
>>> cursor = qs.aggregate(
...     {'$group': {'_id': '$destination',
...                 'price': {'$min': '$price'}}},
...     {'$sort': {'price': pymongo.DESCENDING}},
...     allowDiskUse=True)
>>> list(cursor)
[{'_id': 'GRAND CANYON', 'price': 123.12},
 {'_id': 'MUIR WOODS', 'price': '25.31'},
 {'_id': 'BIGGEST BALL OF TWINE', 'price': '0.25'}]
```

all()
Return a QuerySet over all the objects in this QuerySet.

bulk_create(*object_or_objects*, *retrieve=False*, *full_clean=False*)
Save Model instances in bulk.

Parameters

- *object_or_objects*: A list of MongoModel instances or a single instance.
- *retrieve*: Whether to return the saved MongoModel instances. If `False` (the default), only the ids will be returned.
- *full_clean*: Whether to validate each object by calling the *full_clean()* method before saving. This isn't done by default.

Returns A list of ids for the documents saved, or of the *MongoModel* instances themselves if *retrieve* is `True`.

example:

```
>>> vacation_ids = Vacation.objects.bulk_create([
...     Vacation(destination='TOKYO', travel_method='PLANE'),
...     Vacation(destination='ALGIERS', travel_method='PLANE')])
>>> print(vacation_ids)
[ObjectId('578926716e32ab1d6a8dc718'),
 ObjectId('578926716e32ab1d6a8dc719')]
```

collation(*collation*)
Specify a collation to use for string comparisons.

This will override the default collation of the collection.

Parameters

- *collation*: An instance of *~pymongo.collation.Collation* or a `dict` specifying the collation.

count()
Return the number of objects in this QuerySet.

create(***kwargs*)
Save an instance of this QuerySet's Model.

Parameters

- *kwargs*: Keyword arguments specifying the field values for the *MongoModel* instance to be created.

Returns The *MongoModel* instance, after it has been saved.

example:

```
>>> vacation = Vacation.objects.create(
...     destination="ROME",
...     travel_method="PLANE")
>>> print(vacation)
Vacation(destination='ROME', travel_method='PLANE')
>>> print(vacation.pk)
ObjectId('578925ed6e32ab1d6a8dc717')
```

delete()
Delete objects matched by this QuerySet.

Returns The number of documents deleted.

exclude(**fields*)
Exclude specified fields in QuerySet results.

Parameters

- *fields*: MongoDB names of fields to be excluded.

example:

```
>>> list(Vacation.objects.all())
[Vacation(destination='HAWAII', travel_method='BOAT'),
 Vacation(destination='NAPA', travel_method='CAR')]
>>> list(Vacation.objects.exclude('travel_method'))
[Vacatation(destination='HAWAII'), Vacation(destination='NAPA')]
```

first()
Return the first object from this QuerySet.

get(*raw_query*)
Retrieve the object matching the given criteria.

Raises *DoesNotExist* if no object was found. Raises *MultipleObjectsReturned* if multiple objects were found.

Note that these exception types are specific to the model class itself, so that it's possible to differentiate exceptions on the model type:

```
try:
    user = User.objects.get({'_id': user_id})
    profile = UserProfile.objects.get({'user': user.email})
except User.DoesNotExist:
    # Handle User not existing.
    return redirect_to_registration(user_id)
except UserProfile.DoesNotExist:
    # User has not set up their profile.
    return setup_user_profile(user_id)
```

These model-specific exceptions all inherit from exceptions of the same name defined in the *errors* module, so you can catch them all:

```
try:
    user = User.objects.get({'_id': user_id})
    profile = UserProfile.objects.get({'user': user.email})
except errors.DoesNotExist:
    # Either the User or UserProfile does not exist.
    return redirect_to_404(user_id)
```

limit(*limit*)
Limit the number of objects in this QuerySet.

Parameters

- *limit*: The maximum number of documents to return.

next()

only(**fields*)
Include only specified fields in QuerySet results.

This method is chainable and performs a union of the given fields.

Parameters

- *fields*: MongoDB names of fields to be included.

example:

```
>>> list(Vacation.objects.all())
[Vacation(destination='HAWAII', travel_method='BOAT'),
 Vacation(destination='NAPA', travel_method='CAR')]
>>> list(Vacation.objects.only('travel_method'))
[Vacatation(travel_method='BOAT'), Vacation(travel_method='CAR')]
```

order_by(*ordering*)
Set an ordering for this QuerySet.

Parameters

- *ordering*: The sort criteria. This should be a list of 2-tuples consisting of [(field_name, direction)], where "direction" can be one of ASCENDING or DESCENDING.

project(*projection*)
Specify a raw MongoDB projection to use in QuerySet results.

This method overrides any previous projections on this QuerySet, including those created with *only()* and *exclude()*. Unlike these methods, *project* allows projecting out the primary key. However, note that objects that do not have their primary key cannot be re-saved to the database.

Parameters

- *projection*: A MongoDB projection document.

example:

```
>>> Vacation.objects.project({
...     'destination': 1,
...     'flights': {'$elemMatch': {'available': True}}}).first()
Vacation(destination='HAWAII',
         flights=[{'available': True, 'from': 'SFO'}])
```

raw(*raw_query*)
Filter using a raw MongoDB query.

Parameters

- *raw_query*: A raw MongoDB query.

example:

```
>>> list(Vacation.objects.raw({"travel_method": "CAR"}))
[Vacatation(destination='NAPA', travel_method='CAR'),
 Vacatation(destination='GRAND CANYON', travel_method='CAR')]
```

raw_query
The raw query that will be executed by this QuerySet.

select_related(**fields*)
Allow this QuerySet to pre-fetch objects related to the Model.

Parameters

- *fields*: Names of related fields on this model that should be fetched.

skip(*skip*)
Skip over the first number of objects in this QuerySet.

Parameters

- *skip*: The number of documents to skip.

update(*update*, ***kwargs*)
Update the objects in this QuerySet and return the number updated.

Parameters

- *update*: The modifications to apply.
- *kwargs*: (optional) keyword arguments to pass down to `update_many()`.

example:

```
Subscription.objects.raw({"year": 1995}).update(
    {"$set": {"expired": True}},
    upsert=True)
```

values()
Return Python `dict` instances instead of Model instances.

1.1.6 Working with Files

Tools for working with GridFS.

class pymodm.files.**FieldFile**(*instance*, *field*, *file_id*)
Type returned when accessing a *FileField*.

This type is just a thin wrapper around a *File* and can be treated as a file-like object in most places where a *file* is expected.

close()
Close this file.

delete()
Delete this file.

file
The underlying *File* object.

This will open the file if necessary.

open(*mode='rb'*)
Open this file with the specified *mode*.

save(*name*, *content*)
Save this file.

Parameters

- *name*: The name of the file.
- *content*: The file's contents. This can be a file-like object, string, or bytes.

class pymodm.files.**File**(*file*, *name=None*, *metadata=None*)
Wrapper around a Python *file*.

This class may be assigned directly to a *FileField*.

You can use this class with Python's builtin *file* implementation:

```
>>> my_file = File(open('some/path.txt'))
>>> my_object.filefield = my_file
>>> my_object.save()
```

chunks(*chunk_size=261120*)
Read the file and yield chunks of `chunk_size` bytes.

The default chunk size is the same as the default for GridFS.

This method is useful for streaming large files without loading the entire file into memory.

close()
Close the this file.

open(*mode='rb'*)
Open this file or seek to the beginning if already open.

class pymodm.files.**GridFSFile**(*file_id*, *gridfs_bucket*, *file=None*)
Representation of a file stored on GridFS.

Note that GridFS files are read-only. To change a file on GridFS, you can replace the file with a new version:

```
>>> my_object(upload=File(open('somefile.txt'))).save()
>>> my_object.upload.delete()  # Delete the old version.
>>> my_object.upload = File(open('new_version.txt'))
>>> my_object.save()  # Old file is replaced with the new one.
```

delete()
Delete this file from GridFS.

file
The underlying `GridOut` object.

This will open the file if necessary.

class pymodm.files.**GridFSStorage**(*gridfs_bucket*)
Storage class that uses GridFS to store files.

This is the default Storage implementation for `FileField`.

delete(*file_id*)
Delete the file with the given *file_id*.

exists(*file_id*)
Returns `True` if the file with the given *file_id* exists.

open(*file_id*, *mode='rb'*)
Open a file.

Parameters

- *file_id*: The id of the file.
- *mode*: The file mode. Defaults to `rb`. Not all Storage implementations may support different modes.

Returns The *GridFSFile* with the given *file_id*.

Note: Files from GridFS can only be opened in `rb` mode.

---

save(*name*, *content*, *metadata=None*)
Save *content* in a file named *name*.

Parameters

- *name*: The name of the file.
- *content*: A file-like object, string, or bytes.
- *metadata*: Metadata dictionary to be saved with the file.

Returns The id of the saved file.

class pymodm.files.**ImageFieldFile**(*instance*, *field*, *file_id*)
Type returned when accessing a *ImageField*.

This type is very similar to *FieldFile*, except that it provides a few convenience properties for the underlying image.

format
The format of the image as a string.

height
The height of the image in pixels.

image
The underlying image.

width
The width of the image in pixels.

class pymodm.files.**Storage**
Abstract class that defines the API for managing files.

delete(*file_id*)
Delete the file with the given *file_id*.

exists(*file_id*)
Returns `True` if the file with the given *file_id* exists.

open(*file_id*, *mode='rb'*)
Open a file.

Parameters

- *file_id*: The id of the file.
- *mode*: The file mode. Defaults to `rb`. Not all Storage implementations may support different modes.

Returns The *FieldFile* with the given *file_id*.

save(*name*, *content*, *metadata=None*)
Save *content* in a file named *name*.

Parameters

- *name*: The name of the file.
- *content*: A file-like object, string, or bytes.
- *metadata*: Metadata dictionary to be saved with the file.

Returns The id of the saved file.

1.1.7 Context Managers

class pymodm.context_managers.**collection_options**(*model*, *codec_options=None*, *read_preference=None*, *write_concern=None*, *read_concern=None*)
Context manager that changes the collections options for a Model.

Example:

```
with collection_options(
        MyModel,
        read_preference=ReadPreference.SECONDARY):
    # Read objects off of a secondary.
    MyModel.objects.raw(...)
```

Parameters

- *model*: A *MongoModel* class.
- *codec_options*: An instance of `CodecOptions`.
- *read_preference*: A read preference from the `read_preferences` module.

- *write_concern*: An instance of `WriteConcern`.
- *read_concern*: An instance of `ReadConcern`.

class pymodm.context_managers.**no_auto_dereference**(*model*)
Context manager that turns off automatic dereferencing.

Example:

```
>>> some_profile = UserProfile.objects.first()
>>> with no_auto_dereference(UserProfile):
...     some_profile.user
ObjectId('5786cf1d6e32ab419952fce4')
>>> some_profile.user
User(name='Sammy', points=123)
```

Parameters

- *model*: A *MongoModel* class.

class pymodm.context_managers.**switch_collection**(*model*, *collection_name*)
Context manager that changes the active collection for a Model.

Example:

```
with switch_collection(MyModel, "other_collection"):
    ...
```

Parameters

- *model*: A *MongoModel* class.
- *collection_name*: The name of the new collection to use.

class pymodm.context_managers.**switch_connection**(*model*, *connection_alias*)
Context manager that changes the active connection for a Model.

Example:

```
connect('mongodb://.../mainDatabase', alias='main-app')
connect('mongodb://.../backupDatabase', alias='backup')

# 'MyModel' normally writes to 'mainDatabase'. Let's change that.
with switch_connection(MyModel, 'backup'):
    # This goes to 'backupDatabase'.
    MyModel(name='Bilbo').save()
```

Parameters

- *model*: A *MongoModel* class.
- *connection_alias*: A connection alias that was set up earlier via a call to *connect()*.

1.1.8 Errors

Tools and types for Exception handling.

exception pymodm.errors.**ConfigurationError**
Raised when there is a configuration error.

exception pymodm.errors.**InvalidModel**
Raised if a Model definition is invalid.

exception pymodm.errors.**ModelDoesNotExist**
Raised when a reference to a Model cannot be resolved.

exception pymodm.errors.**OperationError**
Raised when an operation cannot be performed.

exception pymodm.errors.**ValidationError**(*message*, ***kwargs*)
Indicates an error while validating data.

A ValidationError may contain a single error, a list of errors, or even a dictionary mapping field names to errors. Any of these cases are acceptable to pass as a "message" to the constructor for ValidationError.

1.2 Getting Started with PyMODM

This document provides a gentle introduction to pymodm and goes over everything you'll need to write your first application.

1.2.1 Installation

You can install pymodm with pip:

```
pip install pymodm
```

Of course, you'll probably want to have a copy of MongoDB itself running, so you can test your app. You can download it for free from www.mongodb.com.

1.2.2 Connecting to MongoDB

Now that we have all the components, let's connect them together. In pymodm, you connect to MongoDB by calling the *connect()* function:

```
from pymodm import connect

# Connect to MongoDB and call the connection "my-app".
connect("mongodb://localhost:27017/myDatabase", alias="my-app")
```

Let's go through what we just did above. First, we imported the *connect()* method from the *connection* module. Then, we established a connection using a MongoDB connection string. A MongoDB connection string always starts with mongodb:// and can include a multitude of connection options. It's important to note that the connection string provided to connect **must** include a database name ("myDatabase" above). This is the database where all data will reside within your PyMODM application by default.

Another thing we did when we called connect is that we provided an *alias* for the connection ("my-app"). Although providing an alias is optional, doing so may come in handy later, if we ever need to refer to the connection by name. This is useful if we want to have models that use different connection options, or if we ever want to switch what connection a model is using.

1.2.3 Defining Models

Now that we have at least one connection to MongoDB open, we're ready to define our model classes. *MongoModel* is the base class for all top-level models, which represent the data we have stored in MongoDB in a convenient object-oriented way.

Basic Models

Typically, the definition of a MongoModel class will include one or more fields and optionally some metadata, encapsulated in an inner class called `Meta`. Take this example:

```
from pymongo.write_concern import WriteConcern

from pymodm import MongoModel, fields

class User(MongoModel):
    email = fields.EmailField(primary_key=True)
    first_name = fields.CharField()
    last_name = fields.CharField()

    class Meta:
        connection_alias = 'my-app'
        write_concern = WriteConcern(j=True)
```

Our model, `User`, represents documents in the `myDatabase.user` collection in MongoDB. A few things to notice here:

- Our `User` model extends *MongoModel*. This means that it will get its own collection in the database. Any class that inherits *directly* from MongoModel always gets its own collection.
- We gave `User` three fields: `first_name`, `last_name`, and `email`. CharField and EmailField always store their values as unicode strings. The `email` field will also validate its contents as an email address.
- In the `Meta` class, we defined a couple pieces of metadata. The `connection_alias` tells the model what connection to use by default. Here, we're using the connection that we defined earlier, which we gave the name of `my-app`. Additionally, we defined the `write_concern` attribute, which tells the Model what write concern to use by default.
- We set `primary_key=True` in the `email` field. This means that this field will be used as the id for documents of this MongoModel class. Note that this field will actually be called `_id` in the database.

See also:

The *fields* module.

See also:

The list of available *metadata attributes*.

Models that Reference Other Models

Sometimes, our models will need to reference other models. In MongoDB, there are a couple approaches to this:

1. We can store the `_id` of the document we want to reference. When we later need the actual document, we can look it up based on this id. If we need to reference multiple documents, we can store these ids in a list.
2. If we don't need to query the referenced documents outside of our reference structure, we might just embed such documents directly inside the documents that reference them. Similarly, if we have multiple documents we need to reference, we can just have a list of these embedded documents.

Let's take a look at a couple examples of some models that reference the `User` model we wrote earlier:

```
from pymodm import EmbeddedMongoModel, MongoModel, fields

class Comment(EmbeddedMongoModel):
    author = fields.ReferenceField(User)
    content = fields.CharField()

class Post(MongoModel):
    title = fields.CharField()
    author = fields.ReferenceField(User)
    revised_on = fields.DateTimeField()
    content = fields.CharField()
    comments = fields.EmbeddedDocumentListField(Comment)
```

Here we've defined two additional model types: `Comment` and `Post`. These two models demonstrate the two approaches discussed earlier: both `Comment` and `Post` have an author, which is a `User` model. The `User` that represents the author in each case is stored among all the other Users in the `myDatabase.user` collection. In `Comment` and `Post` models, we're just storing the `_id` of the `User` in the `author` field. This is actually the same as the User's `email` field, since we set `primary_key=True` for that field earlier.

`Post` gets a little more interesting. In order to support commenting on a `Post`, what we've done is added a `comments` field, which is an *`EmbeddedDocumentListField`*. This represents the second approach we discussed, where `Comment` objects are embedded directly into our `Post` object. The downside to doing this is that it is difficult to query for individual `Comment` objects. The upside is that we won't have to make an additional query to retrieve all the comments associated with a given `Post`.

See also:

Model Relationships Between Documents

Deleted References

Now that we've defined models that reference other model types, we face another challenge: what happens if a `User` object is deleted? If one of our beloved authors decides to quit the commenting/posting scene, what is to become of their comments and posts? `pymodm` gives us a few options:

- Do nothing (this is the default behavior)
- Change fields that reference the deleted object to `None`.
- Cascade the deletes: when a referenced object is deleted, recursively delete all objects that were referencing it.
- Don't allow deleting objects that still have references to them.
- If the deleted object was just one among potentially many other references stored in a list, remove the reference from this list.

In our case for the `Comment` and `Post` objects, let's delete any comments and posts associated with a `User` after they're gone. This would be the changed definition of the `author` field in each case:

```
author = fields.ReferenceField(User, on_delete=ReferenceField.CASCADE)
```

See also:

The *`ReferenceField`* class.

1.2.4 Creating Data

Alright, now that we've defined models for each MongoDB collection our app will use, let's create some documents!

Saving a Single Instance

Here's one way to set up our first User:

```
User('user@email.com', 'Bob', 'Ross').save()
```

Above, we used positional arguments to construct an instance of `User`. Positional arguments are assigned to fields in the order they were defined in the `User` class. We can also use keyword arguments or a mix of positional/keyword arguments to create MongoModel instances, so this would be equivalent:

```
User('user@email.com', last_name='Ross', first_name='Bob').save()
```

Finally, calling *`save()`* on the instance persists it to the database.

Saving Instances in Bulk

We can also save documents to the database in bulk:

```
users = [
    User('user@email.com', 'Bob', 'Ross'),
    User('anotheruser@email.com', 'David', 'Attenborough')
]
User.objects.bulk_create(users)
```

Updating Documents

There are two ways to update documents in MongoDB with `pymodm`:

1. Change instance attributes to be the way we like, then call *`save()`* on the instance.
2. Use the *`update()`* method on the MongoModel's *`QuerySet`*.

Let's say that we have an instance that looks like this:

```
post = Post(author=some_author, content='This is the first post!').save()
```

Now we realize that we forgot to set the `revised_on` date on the post... oops. Let's fix that by setting the attribute directly per option (1) above:

```
import datetime

# Set the revised_on attribute of our Post from earlier.
post.revised_on = datetime.datetime.now()
# Save the revised document.
post.save()
```

Note that we have to call *`save()`* in order to save any changes we've made to a MongoModel. Setting the attribute just changes its value on our local copy of the document.

The above update strategy works well if we just want to change this single document. But what if we wanted to update documents in bulk or take advantage of a particular MongoDB update operator? The second option grants us more flexibility: we can use the *`update()`* method on the MongoModel's *`QuerySet`*:

```
Post.objects.raw({'revised_on': {'$exists': False}}).update(
    {'$set': {'revised_on': datetime.datetime.now()}})
```

We'll discuss QuerySet objects in more detail in the *Accessing Data* section.

1.2.5 Accessing Data

We've seen how to model the data in our database and how to create some documents, so now it's time to query some of this data. Our primary way of getting to our data happens through the *QuerySet* class, which can be accessed through the `objects` attribute on our Model class. Here's how we could list all the Users we have, for example:

```
for user in User.objects.all():
    print(user.first_name + ' ' + user.last_name)
```

We can do the same thing with `Post` objects. Let's narrow our search to posts that were revised within the last month:

```
import datetime

month_ago = datetime.datetime.now() - datetime.timedelta(days=30)

for post in Post.objects.raw({'revised_on': {'$gte': month_ago}}):
    print(post.title + ' by ' + post.author.first_name)
```

See what we did there? We accessed the `first_name` attribute on the `User` object, even though only the id of the User is technically stored in the `author` field on a Post. When we access the data stored in a *ReferenceField*, it is dereferenced automatically. This makes a separate query to the database. If we didn't want that to happen, we would need to use the *no_auto_dereference()* context manager:

```
from pymodm.context_managers import no_auto_dereference

# Turn off automatic dereferencing for fields defined on "Post".
with no_auto_dereference(Post):
    for post in Post.objects.raw({'revised_on': {'$gte': month_ago}}):
        print(post.title + ' by author with id ' + post.author)
```

Querying Model Subclasses

Earlier, we mentioned that every class that inherits *directly* from *MongoModel* gets its own collection in the database. But what about classes that inherit from some other model class?

```
class ImagePost(Post):
    image = fields.ImageField()
```

The above model subclasses the `Post` model we wrote earlier. Because it does not inherit directly from MongoModel, it does *not* have its own collection. Instead, it shares a collection among all the other `Post` objects. However, we are still able to distinguish between different types when querying the database:

```
for image_post in ImagePost.objects.all():
    assert isinstance(image_post, ImagePost)

for post in Post.objects.all():
    if isinstance(post, ImagePost):
        print('image: ' + repr(post.image))
    print('post content: ' + post.content)
```

How does this work? For every model class that allows inheritance, `pymodm` creates another, hidden field called `_cls` that stores the class of the model that the document refers to. This way, models of different types can be collocated in the same collection while preserving type information.

What if we don't want this `_cls` field to be stored in our documents? This is possible by declaring the model to be *final*, which means that it has to inherit directly from MongoModel and cannot be extended:

```
class PageTheme(MongoModel):
    theme_name = fields.CharField()
    background_color = fields.CharField()
    foreground_color = fields.CharField()

    class Meta:
        final = True
```

1.2.6 Advanced: Managers and Custom QuerySets

We can do a lot with just the tools the default *QuerySet* object provides, but sometimes we may find the need for specialized collection-level functionality, or we might want to write a shortcut for a very common query that we're performing on one or more models.

Let's revisit our `Post` model and add a field called `published`. This will tell us whether the Post has been published or not. Most of the time, we'll probably just want to work with those Post objects that have already been published, but it's going to get annoying *fast* if we have to include `{"published": True}` with every query.

```
class Post(MongoModel):
    title = fields.CharField()
    author = fields.ReferenceField(User)
    revised_on = fields.DateTimeField()
    content = fields.CharField()
    comments = fields.EmbeddedDocumentListField(Comment)
    published = fields.BooleanField(default=False)
```

There are two ways we can easily access only those Posts which aren't drafts:

1. Create a new *QuerySet* class that has a method `published` that filters `Post` objects for ones that have been published.
2. Create a new *Manager* class that always creates instances of `QuerySet` that have the filter `{"published": True}` already applied. This would be handy if we *only* ever cared about Posts that have been published.

We'll discuss each approach in turn.

Custom QuerySets

Let's take a look at the first approach, using a custom QuerySet class:

```
from pymodm.queryset import QuerySet

class PublishedPostQuerySet(QuerySet):
    def published(self):
        '''Return all published Posts.'''
        return self.raw({"published": True})
```

Now that we've defined a QuerySet that has the `published` method, we need to hook it up with a *Manager* class so that we can easily use this `QuerySet` type from our model:

```
from pymodm.manager import Manager

# Create the new Manager class.
PublishedPostManager = Manager.from_queryset(PublishedPostQuerySet)

class Post(MongoModel):
    title = fields.CharField()
```

```
    author = fields.ReferenceField(User)
    revised_on = fields.DateTimeField()
    content = fields.CharField()
    comments = fields.EmbeddedDocumentListField(Comment)
    published = fields.BooleanField(default=False)

    # Change the "objects" manager to use our own Manager, which returns
    # instances of PublishedPostQuerySet:
    objects = PublishedPostManager()

# Get all published Posts.
published_posts = Post.objects.published()
```

Custom Managers

Now let's examine the second approach, where all `QuerySet` instances already have their `{"published": True}` query applied.

When we call a QuerySet method from a Manager, as in `Post.objects.all()`, the *`all()`* method is proxied through the `objects` Manager. The first thing the *`Manager`* does in this case is get a QuerySet instance by calling its own *`get_queryset()`* method, then it applies whatever operation was called on the Manager.

What this means for us is that we can override *`get_queryset()`* to do anything we want to this QuerySet instance before it's returned. Any future operations we do with that QuerySet will have these operations already applied.

The first thing we need to do is subclass *`Manager`*:

```
class PostManager(Manager):
    def get_queryset(self):
        # Override get_queryset() to apply our filter, so that any
        # QuerySet method we call through the Manager already has our query
        # applied.
        return super(PostManager, self).get_queryset().raw(
            {"published": True})
```

Then, as before, we add this Manager to their MongoModel:

```
class Post(MongoModel):
    title = fields.CharField()
    author = fields.ReferenceField(User)
    revised_on = fields.DateTimeField()
    content = fields.CharField()
    comments = fields.EmbeddedDocumentListField(Comment)
    published = fields.BooleanField(default=False)

    # Change the "objects" manager to use our own PostManager.
    objects = PostManager()

# Get all published Posts.
published_posts = Post.objects.all()
```

Of course, we can add whatever other methods we wish to our custom Manager, and they don't all have to return QuerySets. For example, we might define a Manager method to do some complex aggregation:

```
from collections import OrderedDict

class PostManager(Manager):
    def get_queryset(self):
```

```
        # Override get_queryset() to apply our filter, so that any
        # QuerySet method we call through the Manager already has our query
        # applied.
        return super(PostManager, self).get_queryset().raw(
            {"published": True})

    def comment_counts(self):
        '''Get a map of title -> # comments for each Post.'''
        aggregates = self.model.objects.aggregate(
            {'$project': {'title': 1, 'comments': {'$size': '$comments'}}},
            {'$sort': {'comments': -1}}
        )
        return OrderedDict((agg['title'], agg['comments'])
                           for agg in aggregates)
```

Now we can see easily what Posts have the most comments:

```
>>> comment_counts = Post.objects.comment_counts()
>>> print(comment_counts)
OrderedDict([
  ('Getting Started with PyMODM', 9237),
  ('Custom QuerySets and Managers', 423)
])
```

1.2.7 What's Next?

Congratulations! You've read through the Getting Started guide and understand the basics of writing an application using PyMODM. For a more detailed reference of tools that come with PyMODM, check out the *API documentation*.

1.3 Changelog

1.3.1 Version 0.3.0

This release fixes a couple problems from the previous 0.2 release and adds a number of new features, including:

- Support for collations in MongoDB 3.4
- Add a `pymodm.queryset.QuerySet.project()` method to `pymodm.queryset.QuerySet`.
- Allow `DateTimeField` to parse POSIX timestamps (i.e. seconds from the epoch).
- Fix explicit validation of blank fields.

For full list of the issues resolved in this release, visit https://jira.mongodb.org/browse/PYMODM/fixforversion/17662.

1.3.2 Version 0.2.0

This version fixes a few issues and allows defining indexes inside the *Meta* class in a model.

For a complete list of the issues resolved in this release, visit https://jira.mongodb.org/browse/PYMODM/fixforversion/17609.

1.3.3 Version 0.1.0

This version is the very first release of PyMODM.

CHAPTER 2

Changes

See the Changelog for a full list of changes to PyMODM.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

W