OmMongo Documentation

Release 1.0

Bapakode Open Source

Jun 10, 2018

Contents

CHAPTER 1

Contents:

1.1 A Quick OmMongo Introduction

Note: This tutorial is incomplete, but hopefully has enough detail that the places to look in the API documentation for the rest is clear.

In order to use OmMongo to interact with a Mongo database, two things are needed:

- **a *Session* object to handle queuing,** sending, and receiving data from the database.
- **a subclass of *Document* to define a mapping** between a mongo object and python class. The class serves as the mapping, and instances of the class are the python classes that are saved or loaded into the database

This tutorial is going go through the basics of each of these, as well as querying and updating in a way which is programmatic, type safe, and value checked.

1.1.1 Creating a Mapping Object

```
>>> from ommongo.document import Document
>>> from ommongo.fields import *
>>> class BloodDonor(Document):
...     first_name = StringField()
...     last_name = StringField()
...     age = IntField(min_value=0)
...
...     gender = EnumField(StringField(), 'male', 'female')
...     blood_type = EnumField(StringField(), 'O+','A+','B+','AB+','O-','A-','B-','AB-
↪')
...     def __str__(self):
...         return '%s %s (%s; Age: %d; Type: %s)' % (self.first_name, self.last_name,
...             self.gender, self.age, self.blood_type)
```

The above code creates a class `BloodDoner` which will has all of the information necessary to create python objects to save, load database objects (though there are none at the moment), and to construct queries for these objects.

It also introduces several `field` subclasses which are used to define the key/value pairs in the mongo Document. The *`StringField`* and *`IntField`* are fairly self-explanatory. The *`ommongo.fields.EnumField`* is more complex. It takes a field as its first argument followed by any number of values of the type accepted by that field. The EnumField will check both that the value it is constructed with is of the correct type, but also that its value is one of the values given in the constructor.

See also:

`Document` and *`fields`*

1.1.2 Sessions

```
>>> from ommongo.session import Session
>>> session = Session.connect('ommongo-tutorial')
>>> session.clear_collection(BloodDonor)
```

The above code creates a session object by connecting to a local mongo server and accessing the ommongo-tutorial database. Any arguments after the database name will be used as arguments to pymongo's connection function. It's also possible to directly construct a session using a pymongo database object, allowing one connection to be used for multiple sessions. The last line clears all objects from the collection used for BloodDonor objects, in case this tutorial code had run before.

```
>>> donor = BloodDonor(first_name='Jeff', last_name='Jenkins',
...                    age=28, blood_type='O+', gender='male')
>>> session.save(donor)
```

When `save` is called on the `donor` object the session serializes it into a form that the Mongo database understands, and then inserts it. Once the save command is complete the `_id` field of `donor` object is set.

To make the next section more interesting more objects are needed, so we'll add some of the cast of *Community*:

```
>>> session.save(BloodDonor(first_name='Jeff', last_name='Winger', age=38, blood_type=
↪'O+', gender='male'))
>>> session.save(BloodDonor(first_name='Britta', last_name='Perry', age=27, blood_
↪type='A+', gender='female'))
>>> session.save(BloodDonor(first_name='Abed', last_name='Nadir', age=29, blood_type=
↪'O+', gender='male'))
>>> session.save(BloodDonor(first_name='Shirley', last_name='Bennett', age=39, blood_
↪type='O-', gender='female'))
```

1.1.3 Querying

```
>>> for donor in session.query(BloodDonor).filter(BloodDonor.first_name == 'Jeff'):
>>>     print donor
Jeff Jenkins (male; Age: 28; Type: O+)
Jeff Winger (male; Age: 38; Type: O+)
```

The above code uses the *`query()`* method of the `session` object to start a query on the BloodDonor collection. The filter function on a query object allows constraints to be added to the results returned. In this case all donors who have the name Jeff are being printed. The attributes of a Document subclass are used to access the names of fields in such a way that they generate query expressions which `filter` can use.

Multiple filters can be applied by chaining calls to filter or by adding comma-separated query expressions inside a single call. The following two examples return the same results:

```
>>> query = session.query(BloodDonor)
>>> for donor in query.filter(BloodDonor.first_name == 'Jeff', BloodDonor.age < 30):
>>>     print donor
Jeff Jenkins (male; Age: 28; Type: O+)
```

```
>>> query = session.query(BloodDonor)
>>> for donor in query.filter(BloodDonor.first_name == 'Jeff').filter(BloodDonor.age
↪< 30):
>>>     print donor
Jeff Jenkins (male; Age: 28; Type: O+)
```

Instead of getting elements by iterating on the query, the *one()* and *first()* methods can be used. `one` returns the first result and raises an exception if there is not exactly one returned result. `first` returns the first result if there is one, and `None` if there is not one.

See also:

Mongo Query Expression Language for all available types of query expressions

1.1.4 Updating

There are a number of methods available for updating results of a query (rather than updating results by loading an object and re-inserting it). Here's an example where my age changed and it turned out I had the wrong blood type:

```
>>> query = session.query(BloodDonor).filter(BloodDonor.first_name == 'Jeff',␣
↪BloodDonor.last_name == 'Jenkins')
>>> query.inc(BloodDonor.age, 1).set(BloodDonor.blood_type, 'O-').execute()
>>> query.one()
Jeff Jenkins (male; Age: 29; Type: O-)
```

inc() and *set()* are two of the methods which can be used to do updates on a query. Once an update method has been called on a query any further chaining is on a *UpdateExpression*, so no further filtering of results will be possible. The *execute()* method causes the update to actually happen in the database.

It is also possible to call *update()* on the session to automatically update a particular object with an update command instead of either saving the object again (which is synchronous) or constructing the update expression yourself as above.

By default all fields use the set operation to update their values, but that can be overriden by passing the on_update operator to the field constructor, or a keyword argument to update. The value of these arguments is currently a string with the mongo operation you want to do instead. For example, `IntField(on_update='$inc')` is an IntField which will increment when update is called. `Session.update` *is still experimental, so watch out for side-effects*

See also:

- *Update Expressions* for all available types of update expressions
- *ommongo.session.Session.update()* for details of automatic updates

1.1.5 Performance

Note: still to come!

- Indexes (For now, see *Index*)
- Hints (For now, see *hint_asc()* and *hint_desc()*)
- Partial Document Loading (For now, see the `fields` parameter to *Document*)

1.2 MongoAlchemy in-depth Tutorials

1.2.1 References Tutorial (Coming Soon)

1.2.2 Transactions and Caching Tutorial (Coming Soon)

1.2.3 Query Language Tutorial (Coming Soon)

1.2.4 Update Expressions Tutorial (Coming Soon)

1.2.5 Custom Field Tutorial (Coming Soon)

1.3 API documentation

Contents:

1.3.1 Session

Session objects handles the actual queueing of database operations. The primary methods on a session are query, save, and flush.

The session also responsible for ordering operations and knowing when operations need to be flushed, although it does not currently do anything intelligent for ordering.

class ommongo.session.**Session**(*database*, *tz_aware=False*, *timezone=None*, *safe=False*, *cache_size=0*, *auto_ensure=True*)

Create a session connecting to *database*.

Parameters

- **database** – the database to connect to. Should be an instance of `pymongo.database.Database`
- **safe** – Whether the "safe" option should be used on mongo writes, blocking to make sure there are no errors.
- **auto_ensure** – Whether to implicitly call ensure_indexes on all write operations.

Fields:

- db: the underlying pymongo database object
- queue: the queue of unflushed database commands (currently useless since there aren't any operations which defer flushing)
- cache_size: The size of the identity map to keep. When objects are pulled from the DB they are checked against this map and if present, the existing object is used. Defaults to 0, use None to only clear at session end.

add(*item*, *safe=None*)
Add an item into the queue of things to be inserted. Does not flush.

add_to_session(*obj*)

auto_ensure_indexes(*cls*)

begin_trans()

clear_cache()

clear_collection(**classes*)
Clear all objects from the collections associated with the objects in **cls*. **use with caution!**

clear_index(*collection*)

clear_queue(*trans_id=None*)
Clear the queue of database operations without executing any of the pending operations

clone(*document*)
Serialize a document, remove its _id, and deserialize as a new object

classmethod connect(*database*, *timezone=None*, *cache_size=0*, *auto_ensure=True*, *replica_set=None*, **args*, ***kwds*)
connect is a thin wrapper around __init__ which creates the database connection that the session will use.

Parameters

- **database** – the database name to use. Should be an instance of `basestring`
- **safe** – The value for the "safe" parameter of the Session init function
- **auto_ensure** – Whether to implicitly call ensure_indexes on all write operations.
- **replica_set** – The replica-set to use (as a string). If specified, `pymongo.mongo_replica_set_client.MongoReplicaSetClient` is used instead of `pymongo.mongo_client.MongoClient`
- **args** – arguments for `pymongo.mongo_client.MongoClient`
- **kwds** – keyword arguments for `pymongo.mongo_client.MongoClient`

dereference(*ref*, *allow_none=False*)

end()
End the session. Flush all pending operations and ending the *pymongo* request

end_trans(*exc_type=None*, *exc_val=None*, *exc_tb=None*)

ensure_indexes(*cls*)

execute_query(*query*, *session*)
Get the results of `query`. This method does flush in a transaction, so any objects retrieved which are not in the cache which would be updated when the transaction finishes will be stale

flush(*safe=None*)
Perform all database operations currently in the queue

get_indexes(*cls*)
Get the index information for the collection associated with *cls*. Index information is returned in the same format as *pymongo*.

insert(*item*, *safe=None*)
[DEPRECATED] Please use save() instead. This actually calls the underlying save function, so the name is confusing.

Insert an item into the work queue and flushes.

query(*type*, *exclude_subclasses=False*)

Begin a query on the database's collection for *type*. If *type* is an instance of basesting, the query will be in raw query mode which will not check field values or transform returned results into python objects.

See also:

Query class

refresh(*document*)
Load a new copy of a document from the database. does not replace the old one

remove(*obj*, *safe=None*)
Remove a particular object from the database. If the object has no mongo ID set, the method just returns. If this is a partial document without the mongo ID field retrieved a `FieldNotRetrieved` will be raised

Parameters

- **obj** – the object to save
- **safe** – whether to wait for the operation to complete. Defaults to the session's `safe` value.

remove_query(*type*)
Begin a remove query on the database's collection for *type*.

See also:

`RemoveQuery` class

save(*item*, *safe=None*)
Saves an item into the work queue and flushes.

update(*item*, *id_expression=None*, *upsert=False*, *update_ops={}*, *safe=None*, ***kwargs*)
Update an item in the database. Uses the on_update keyword to each field to decide which operations to do, or.

Parameters

- **item** – An instance of a *Document* subclass
- **id_expression** – A query expression that uniquely picks out the item which should be updated. If id_expression is not passed, update uses item.mongo_id.
- **upsert** – Whether the update operation should be an upsert. If the item may not be in the database yet this should be True
- **update_ops** – By default the operation used to update a field is specified with the on_update argument to its constructor. To override that value, use this dictionary, with `QueryField` objects as the keys and the mongo operation to use as the values.
- **kwargs** – The kwargs are merged into update_ops dict to decide which fields to update the operation for. These can only be for the top-level document since the keys are just strings.

> **Warning:** This operation is **experimental** and **not fully tested**, although it does have code coverage.

1.3.2 Schema — Document-Object Mapper and Schema Definitions

Modules:

Documents and Indexes

exception ommongo.document.**BadIndexException**

class ommongo.document.**DictDoc**

Adds a mapping interface to a document. Supports __getitem__ and __contains__. Both methods will only retrieve values assigned to a field, not methods or other attributes.

setdefault(*name*, *value*)

if the name is set, return its value. Otherwse set name to value and return value

class ommongo.document.**Document**(*retrieved_fields=None*, *loading_from_db=False*, ***kwargs*)

Parameters

- **retrieved_fields** – The names of the fields returned when loading a partial object. This argument should not be explicitly set by subclasses
- ****kwargs** – The values for all of the fields in the document. Any additional fields will raise a ExtraValueException and any missing (but required) fields will raise a MissingValueException. Both types of exceptions are subclasses of DocumentException.

classmethod add_subclass(*subclass*)

Register a subclass of this class. Maps the subclass to the value of subclass.config_polymorphic_identity if available.

classmethod base_query(*exclude_subclasses=False*)

Return the base query for this kind of document. If this class is not polymorphic, the query is empty. If it is polymorphic then a filter is added to match only this class and its subclasses.

Parameters **exclude_subclasses** – If this is true, only match the current class. If it is false, the default, also return subclasses of this class.

classmethod class_name()

Returns the name of the class. The name of the class is also the default collection name.

See also:

get_collection_name()

config_default_sort = None

config_extra_fields = 'error'

config_full_name = None

config_namespace = 'global'

config_polymorphic = None

config_polymorphic_collection = False

config_polymorphic_identity = None

classmethod get_collection_name()

Returns the collection name used by the class. If the config_collection_name attribute is set it is used, otherwise the name of the class is used.

get_dirty_ops(*with_required=False*)

Returns a dict with the update operations necessary to make the changes to this object to the database version. It is mainly used internally for *update()* but may be useful for diagnostic purposes as well.

Parameters **with_required** – Also include any field which is required. This is useful if the method is being called for the purposes of an upsert where all required fields must always be sent.

get_extra_fields()
if *Document.config_extra_fields* is set to 'ignore', this method will return a dictionary of the fields which couldn't be mapped to the document.

classmethod get_fields()
Returns a dict mapping the names of the fields in a document or subclass to the associated *Field*

get_index_score()
Get index scores from full-text search.

classmethod get_indexes()
Returns all of the *Index* instances for the current class.

classmethod get_subclass(*obj*)
Get the subclass to use when instantiating a polymorphic object. The default implementation looks at `cls.config_polymorphic` to get the name of an attribute. Subclasses automatically register their value for that attribute on creation via their `config_polymorphic_identity` field. This process is then repeated recursively until None is returned (indicating that the current class is the correct one)

This method can be overridden to allow any method you would like to use to select subclasses. It should return either the subclass to use or None, if the original class should be used.

has_id()

mongo_id

to_ref(*db=None*)

classmethod transform_incoming(*obj*, *session*)
Tranform the SON object into one which will be able to be unwrapped by this document class.

This method is designed for schema migration systems.

class ommongo.document.**DocumentMeta**

mro() → list
return a type's method resolution order

class ommongo.document.**Index**
This class is used in the class definition of a *Document* to specify a single, possibly compound, index. pymongo's `ensure_index` will be called on each index before a database operation is executed on the owner document class.

Example

```
>>> class Donor(Document):
...     name = StringField()
...     age = IntField(min_value=0)
...     blood_type = StringField()
...
...     i_name = Index().ascending('name')
...     type_age = Index().ascending('blood_type').descending('age')
```

ASCENDING = 1

DESCENDING = -1

ascending(*name*)
Add a descending index for `name` to this index.

Parameters **name** – Name to be used in the index

descending(*name*)
Add a descending index for `name` to this index.

Parameters **name** – Name to be used in the index

ensure(*collection*)
Call the pymongo method `ensure_index` on the passed collection.

Parameters **collection** – the `pymongo` collection to ensure this index is on

expire(*after*)
Add an expire after option to the index

Param after: Number of second before expiration

geo2d(*name*, *min=None*, *max=None*)
Create a 2d index. See: http://www.mongodb.org/display/DOCS/Geospatial+Indexing

Parameters

- **name** – Name of the indexed column
- **min** – minimum value for the index
- **max** – minimum value for the index

geo_haystack(*name*, *bucket_size*)
Create a Haystack index. See: http://www.mongodb.org/display/DOCS/Geospatial+Haystack+Indexing

Parameters

- **name** – Name of the indexed column
- **bucket_size** – Size of the haystack buckets (see mongo docs)

unique(*drop_dups=False*)
Make this index unique, optionally dropping duplicate entries.

Parameters **drop_dups** – Drop duplicate objects while creating the unique index? Default to `False`

class ommongo.document.**Value**(*field*, *document*, *from_db=False*, *extra=False*, *retrieved=True*)

clear_dirty()

delete()

Document

class ommongo.document.**Document**(*retrieved_fields=None*, *loading_from_db=False*, ***kwargs*)

Parameters

- **retrieved_fields** – The names of the fields returned when loading a partial object. This argument should not be explicitly set by subclasses
- ****kwargs** – The values for all of the fields in the document. Any additional fields will raise a `ExtraValueException` and any missing (but required) fields

will raise a `MissingValueException`. Both types of exceptions are subclasses of `DocumentException`.

classmethod add_subclass(*subclass*)
Register a subclass of this class. Maps the subclass to the value of subclass.config_polymorphic_identity if available.

classmethod base_query(*exclude_subclasses=False*)
Return the base query for this kind of document. If this class is not polymorphic, the query is empty. If it is polymorphic then a filter is added to match only this class and its subclasses.

Parameters **exclude_subclasses** – If this is true, only match the current class. If it is false, the default, also return subclasses of this class.

classmethod class_name()
Returns the name of the class. The name of the class is also the default collection name.

See also:

get_collection_name()

config_default_sort = None

config_extra_fields = 'error'

config_full_name = None

config_namespace = 'global'

config_polymorphic = None

config_polymorphic_collection = False

config_polymorphic_identity = None

classmethod get_collection_name()
Returns the collection name used by the class. If the `config_collection_name` attribute is set it is used, otherwise the name of the class is used.

get_dirty_ops(*with_required=False*)
Returns a dict with the update operations necessary to make the changes to this object to the database version. It is mainly used internally for *update()* but may be useful for diagnostic purposes as well.

Parameters **with_required** – Also include any field which is required. This is useful if the method is being called for the purposes of an upsert where all required fields must always be sent.

get_extra_fields()
if *Document.config_extra_fields* is set to 'ignore', this method will return a dictionary of the fields which couldn't be mapped to the document.

classmethod get_fields()
Returns a dict mapping the names of the fields in a document or subclass to the associated *Field*

get_index_score()
Get index scores from full-text search.

classmethod get_indexes()
Returns all of the *Index* instances for the current class.

classmethod get_subclass(*obj*)
Get the subclass to use when instantiating a polymorphic object. The default implementation looks at `cls.config_polymorphic` to get the name of an attribute. Subclasses automatically register their

value for that attribute on creation via their `config_polymorphic_identity` field. This process is then repeated recursively until None is returned (indicating that the current class is the correct one)

This method can be overridden to allow any method you would like to use to select subclasses. It should return either the subclass to use or None, if the original class should be used.

has_id()

mongo_id

to_ref(*db=None*)

classmethod transform_incoming(*obj*, *session*)
Tranform the SON object into one which will be able to be unwrapped by this document class.

This method is designed for schema migration systems.

Index

class ommongo.document.**Index**
This class is used in the class definition of a *Document* to specify a single, possibly compound, index. pymongo's `ensure_index` will be called on each index before a database operation is executed on the owner document class.

Example

```
>>> class Donor(Document):
...     name = StringField()
...     age = IntField(min_value=0)
...     blood_type = StringField()
...
...     i_name = Index().ascending('name')
...     type_age = Index().ascending('blood_type').descending('age')
```

ASCENDING = 1

DESCENDING = -1

ascending(*name*)
Add a descending index for `name` to this index.

Parameters **name** – Name to be used in the index

descending(*name*)
Add a descending index for `name` to this index.

Parameters **name** – Name to be used in the index

ensure(*collection*)
Call the pymongo method `ensure_index` on the passed collection.

Parameters **collection** – the `pymongo` collection to ensure this index is on

expire(*after*)
Add an expire after option to the index

Param after: Number of second before expiration

geo2d(*name*, *min=None*, *max=None*)
Create a 2d index. See: http://www.mongodb.org/display/DOCS/Geospatial+Indexing

Parameters

- **name** – Name of the indexed column
- **min** – minimum value for the index
- **max** – minimum value for the index

geo_haystack(*name*, *bucket_size*)
Create a Haystack index. See: http://www.mongodb.org/display/DOCS/Geospatial+Haystack+Indexing

Parameters

- **name** – Name of the indexed column
- **bucket_size** – Size of the haystack buckets (see mongo docs)

unique(*drop_dups=False*)
Make this index unique, optionally dropping duplicate entries.

Parameters **drop_dups** – Drop duplicate objects while creating the unique index? Default to `False`

Field Types

Field

class ommongo.fields.**Field**(*required=True*, *default=UNSET*, *default_f=None*, *db_field=None*, *allow_none=False*, *on_update='$set'*, *validator=None*, *unwrap_validator=None*, *wrap_validator=None*, *_id=False*, *proxy=None*, *iproxy=None*, *ignore_missing=False*)

Parameters

- **required** – The field must be passed when constructing a document (optional. default: `True`)
- **default** – Default value to use if one is not given (optional.)
- **db_field** – name to use when saving or loading this field from the database (optional. default is the name the field is assigned to on a documet)
- **allow_none** – allow `None` as a value (optional. default: False)
- **validator** – a callable which will be called on objects when wrapping/unwrapping
- **unwrap_validator** – a callable which will be called on objects when unwrapping
- **wrap_validator** – a callable which will be called on objects when wrapping
- **_id** – Set the db_field to _id. If a field has this the "mongo_id" field will also be removed from the document the field is on.

The general validator is called after the field's validator, but before either of the wrap/unwrap versions. The validator should raise a BadValueException if it fails, but if it returns False the field will raise an exception with a generic message.

is_sequence_field = False

Primitive Fields

class ommongo.fields.**StringField**(*max_length=None*, *min_length=None*, ***kwargs*)
Unicode Strings. `unicode` is used to wrap and unwrap values, and any subclass of basestring is an acceptable input

Parameters

- **max_length** – maximum string length
- **min_length** – minimum string length
- **kwargs** – arguments for *Field*

is_sequence_field = False

class ommongo.fields.**BoolField**(***kwargs*)
`True` or `False`.

is_sequence_field = False

class ommongo.fields.**NumberField**(*constructor*, *min_value=None*, *max_value=None*, ***kwargs*)
Base class for numeric fields

Parameters

- **max_value** – maximum value
- **min_value** – minimum value
- **kwargs** – arguments for *Field*

is_sequence_field = False

class ommongo.fields.**IntField**(***kwargs*)
Subclass of *NumberField* for `int`

Parameters

- **max_value** – maximum value
- **min_value** – minimum value
- **kwargs** – arguments for *Field*

is_sequence_field = False

class ommongo.fields.**FloatField**(***kwargs*)
Subclass of *NumberField* for `float`

Parameters

- **max_value** – maximum value
- **min_value** – minimum value
- **kwargs** – arguments for *Field*

is_sequence_field = False

class ommongo.fields.**TupleField**(**item_types*, ***kwargs*)
Represents a field which is a tuple of a fixed size with specific types for each element in the field.

Examples `TupleField(IntField(), BoolField())` would accept `[19, False]` as a value for both wrapping and unwrapping.

Parameters

- **item_types** – instances of *Field*, in the order they will appear in the tuples.
- **kwargs** – arguments for *Field*

is_sequence_field = False

class ommongo.fields.**EnumField**(*item_type*, **values*, ***kwargs*)
Represents a single value out of a list of possible values, all of the same type. == is used for comparison

Example: `EnumField(IntField(), 4, 6, 7)` would accept anything in `(4, 6, 7)` as a value. It would not accept `5`.

Parameters

- **item_type** – Instance of *Field* to use for validation, and (un)wrapping
- **values** – Possible values. `item_type.is_valid_wrap(value)` should be `True`

is_sequence_field = False

class ommongo.fields.**ObjectIdField**(*session=None*, *auto=False*, ***kwargs*)
pymongo Object ID object. Currently this is probably too strict. A string version of an ObjectId should also be acceptable

gen()
Helper method to create a new ObjectId

is_sequence_field = False

class ommongo.fields.**AnythingField**(*required=True*, *default=UNSET*, *default_f=None*, *db_field=None*, *allow_none=False*, *on_update='$set'*, *validator=None*, *unwrap_validator=None*, *wrap_validator=None*, *_id=False*, *proxy=None*, *iproxy=None*, *ignore_missing=False*)
A field that passes through whatever is set with no validation. Useful for free-form objects

Parameters

- **required** – The field must be passed when constructing a document (optional. default: `True`)
- **default** – Default value to use if one is not given (optional.)
- **db_field** – name to use when saving or loading this field from the database (optional. default is the name the field is assigned to on a documet)
- **allow_none** – allow `None` as a value (optional. default: False)
- **validator** – a callable which will be called on objects when wrapping/unwrapping
- **unwrap_validator** – a callable which will be called on objects when unwrapping
- **wrap_validator** – a callable which will be called on objects when wrapping
- **_id** – Set the db_field to _id. If a field has this the "mongo_id" field will also be removed from the document the field is on.

The general validator is called after the field's validator, but before either of the wrap/unwrap versions. The validator should raise a BadValueException if it fails, but if it returns False the field will raise an exception with a generic message.

is_sequence_field = False

Date and Time Fields

class ommongo.fields.**DateTimeField**(*min_date=None*, *max_date=None*, *use_tz=False*, ***kwargs*)
Field for datetime objects.

Parameters

- **max_date** – maximum date
- **min_date** – minimum date
- **use_tz** – Require a timezone-aware datetime (via pytz). Values are converted to UTC before saving. min and max dates are currently ignored when use_tz is on. You MUST pass a timezone into the session
- **kwargs** – arguments for *Field*

is_sequence_field = False

ommongo.fields.**CreatedField**(*name='created'*, *tz_aware=False*, ***kwargs*)

A shortcut field for creation time. It sets the current date and time when it enters the database and then doesn't update on further saves.

If you've used the Django ORM, this is the equivalent of auto_now_add

Parameters **tz_aware** – If this is True, the value will be returned in the local time of the session. It is always saved in UTC

ommongo.fields.**ModifiedField**(*tz_aware=False*, ***kwargs*)

A shortcut field for modified time. It sets the current date and time when it enters the database and then updates when the document is saved or updated

If you've used the Django ORM, this is the equivalent of auto_now

WARNINGS: When this field's parent object is sent to the database its modified time is set. The local copy is not updated for technical reasons. Hopefully this will not be the case in the future.

Parameters **tz_aware** – If this is True, the value will be returned in the local time of the session. It is always saved in UTC

Sequence Type Fields

class ommongo.fields.**ListField**(*item_type*, ***kwargs*)

Field representing a python list.

Parameters

- **item_type** – *Field* instance used for validation and (un)wrapping
- **min_capacity** – minimum number of items contained in values
- **max_capacity** – maximum number of items contained in values
- **default_empty** – the default is an empty sequence.

is_sequence_field = True

rel(*ignore_missing=False*)

class ommongo.fields.**SetField**(*item_type*, ***kwargs*)

Field representing a python set.

Parameters

- **item_type** – *Field* instance used for validation and (un)wrapping
- **min_capacity** – minimum number of items contained in values
- **max_capacity** – maximum number of items contained in values
- **default_empty** – the default is an empty sequence.

is_sequence_field = True

rel(*ignore_missing=False*)

Mapping Type Fields

class ommongo.fields.**DictField**(*value_type*, *default_empty=False*, ***kwargs*)
Stores String to `value_type` Dictionaries. For non-string keys use *KVField*. Strings also must obey the mongo key rules (no `.` or `$`)

Parameters **value_type** – the Field type to use for the values

is_sequence_field = False

class ommongo.fields.**KVField**(*key_type*, *value_type*, *default_empty=False*, ***kwargs*)
Like a DictField, except it allows arbitrary keys. The DB Format for a `KVField` is `[ { 'k' : key, 'v' : value }, ...]`. Queries on keys and values. can be done with `.k` and `.v`

Parameters

- **key_type** – the Field type to use for the keys
- **value_type** – the Field type to use for the values

is_sequence_field = False

Document Field

class ommongo.fields.**DocumentField**(*document_class*, ***kwargs*)
A field which wraps a `Document`

is_sequence_field = False

Reference Field

class ommongo.fields.**RefField**(*type=None*, *db=None*, *db_required=False*, *namespace='global'*, ***kwargs*)
A ref field wraps a mongo DBReference. It DOES NOT currently handle saving the referenced object or updates to it, but it can handle auto-loading.

Parameters

- **type** – (optional) the Field type to use for the values. It must be a DocumentField. If you want to save refs to raw mongo objects, you can leave this field out
- **db** – (optional) The database to load the object from. Defaults to the same database as the object this field is bound to.
- **namespace** – If using the namespace system and using a collection name instead of a type, selects which namespace to use

dereference(*session*, *ref*, *allow_none=False*)
Dereference a pymongo "DBRef" to this field's underlying type

is_sequence_field = False

rel(*allow_none=False*)
Used to create an attribute which will auto-dereference a RefField or SRefField.

Example:

```
employer_ref = SRefField(Employer)
employer = employer_ref.rel()
```

class ommongo.fields.**SRefField**(*type*, *db=None*, ***kwargs*)

A Simple RefField (SRefField) looks like an ObjectIdField in the database, but acts like a mongo DBRef. It uses the passed in type to determine where to look for the object (and assumes the current database).

dereference(*session*, *ref*, *allow_none=False*)

Dereference an ObjectID to this field's underlying type

is_sequence_field = False

rel(*allow_none=False*)

Used to create an attribute which will auto-dereference a RefField or SRefField.

Example:

```
employer_ref = SRefField(Employer)
employer = employer_ref.rel()
```

Computed Field

class ommongo.fields.**ComputedField**(*computed_type*, *fun*, *one_time=False*, *deps=None*, ***kwargs*)

A computed field is generated based on an object's other values. It will generally be created with the @computed_field decorator, but can be passed an arbitrary function.

The function should take a dict which will contains keys with the names of the dependencies mapped to their values.

The computed value is recalculated every the field is accessed unless the one_time field is set to True.

Example:

```
>>> class SomeDoc(Document):
...     @computed_field
...     def last_modified(obj):
...         return datetime.datetime.utcnow()
```

Warning: The computed field interacts in an undefined way with partially loaded documents right now. If using this class watch out for strange behaviour.

Parameters

- **fun** – the function to compute the value of the computed field
- **computed_type** – the type to use when wrapping the computed field
- **deps** – the names of fields on the current object which should be passed in to compute the value

is_sequence_field = False

ommongo.fields.**computed_field**(*computed_type*, *deps=None*, ***kwargs*)

1.3.3 Expression Language — Querying and Updating

Query Objects

class ommongo.query.**Query**(*type*, *session*, *exclude_subclasses=False*)

A query object has all of the methods necessary to programmatically generate a mongo query as well as methods to retrieve results of the query or do an update based on it.

In general a query object should be created via `Session.query`, not directly.

Parameters

- **type** – A subclass of class:*ommongo.document.Document*
- **db** – The *Session* which this query is associated with.
- **exclude_subclasses** – If this is set to false (the default) and *type* is polymorphic then subclasses are also retrieved.

add_to_set(*qfield*, *value*)

Refer to: *add_to_set()*

aggregate(*raw_query*)

all()

Return all of the results of a query in a list

append(*qfield*, *value*)

Refer to: *append()*

ascending(*qfield*)

Sort the result based on `qfield` in ascending order. These calls can be chained to sort by multiple fields.

Parameters **qfield** – Instance of :class:`ommongo.query.QueryField` specifying which field to sort by.

clone()

Creates a clone of the current query and all settings. Further updates to the cloned object or the original object will not affect each other

count(*with_limit_and_skip=False*)

Execute a count on the number of results this query would return.

Parameters **with_limit_and_skip** – Include `.limit()` and `.skip()` arguments in the count?

descending(*qfield*)

Sort the result based on `qfield` in ascending order. These calls can be chained to sort by multiple fields.

Parameters **qfield** – Instance of :class:`ommongo.query.QueryField` specifying which field to sort by.

distinct(*key*)

Execute this query and return all of the unique values of `key`.

Parameters **key** – the instance of `ommongo.QueryField` to use as the distinct key.

explain()

Executes an explain operation on the database for the current query and returns the raw explain object returned.

extend(*qfield*, **value*)

Refer to: *extend()*

fields(**fields*)
Only return the specified fields from the object. Accessing a field that was not specified in `fields` will result in a :class:`ommongo.document.FieldNotRetrieved` exception being raised

Parameters **fields** – Instances of :class:`ommongo.query.QueryField` specifying which fields to return

filter(**query_expressions*)
Apply the given query expressions to this query object

Example: `s.query(SomeObj).filter(SomeObj.age > 10, SomeObj.blood_type == 'O')`

Parameters **query_expressions** – Instances of *ommongo.query_expression.QueryExpression*

See also:

QueryExpression class

filter_by(***filters*)
Filter for the names in `filters` being equal to the associated values. Cannot be used for sub-objects since keys must be strings

filter_dict(*query*, ***kwargs*)
Filter for `DictField()`.

Examples: `query.filter_dict({"User.Fullname": "Oji"})`

filter_like(***filters*)
Filter query using re.compile().

Examples: `query.filter_like(Name="andi")`

find_and_modify(*new=False*, *remove=False*)
The mongo "find and modify" command. Behaves like an update expression in that "execute" must be called to do the update and return the results.

Parameters

- **new** – Whether to return the new object or old (default: False)
- **remove** – Whether to remove the object before returning it

first()
Execute the query and return the first result. Unlike `one`, if there are multiple documents it simply returns the first one. If there are no documents, first returns `None`

hint_asc(*qfield*)
Applies a hint for the query that it should use a (`qfield`, ASCENDING) index when performing the query.

Parameters **qfield** – the instance of `ommongo.QueryField` to use as the key.

hint_desc(*qfield*)
Applies a hint for the query that it should use a (`qfield`, DESCENDING) index when performing the query.

Parameters **qfield** – the instance of `ommongo.QueryField` to use as the key.

in_(*qfield*, **values*)
Check to see that the value of `qfield` is one of `values`

Parameters

- **qfield** – Instances of *ommongo.query_expression.QueryExpression*
- **values** – Values should be python values which `qfield` understands

inc(**args*, ***kwargs*)
Refer to: *inc()*

limit(*limit*)
Sets the limit on the number of documents returned

Parameters **limit** – the number of documents to return

map_reduce(*mapper*, *reducer*, *key*, *query*)

nin(*qfield*, **values*)
Check to see that the value of `qfield` is not one of `values`

Parameters

- **qfield** – Instances of *ommongo.query_expression.QueryExpression*
- **values** – Values should be python values which `qfield` understands

not_(**query_expressions*)
Add a $not expression to the query, negating the query expressions given.

Examples: `query.not_(SomeDocClass.age <= 18)` becomes `{'age' : { '$not' : { '$gt' : 18 } }}`

Parameters **query_expressions** – Instances of *ommongo.query_expression.QueryExpression*

one()
Execute the query and return one result. If more than one result is returned, raises a `BadResultException`

or_(*first_qe*, **qes*)
Add a $not expression to the query, negating the query expressions given. The `|` `operator` on query expressions does the same thing

Examples: `query.or_(SomeDocClass.age == 18, SomeDocClass.age == 17)` becomes `{'$or' : [{ 'age' : 18 }, { 'age' : 17 }]}`

Parameters **query_expressions** – Instances of *ommongo.query_expression.QueryExpression*

pop_first(*qfield*)
Refer to: *pop_first()*

pop_last(*qfield*)
Refer to: *pop_last()*

query
The mongo query object which would be executed if this Query object were used

query_bypass(*query*, *raw_output=True*)
Bypass query meaning that field check and validation is skipped, then query object directly executed by pymongo.

Parameters **raw_output** – Skip OmMongo ORM layer (default: True)

raw_output()
Turns on raw output, meaning that the OmMongo ORM layer is skipped and the results from pymongo are returned. Useful if you want to use the query functionality without getting python objects back

remove(*qfield*, *value*)
Refer to: *remove()*

remove_all(*qfield*, **value*)
Refer to: *remove_all()*

search(*value*, *createIndex=None*)
Full-text support, make sure that text index already exist on collection. Raise IndexNotFound if text index not exist.

Examples: `query.search('pecel lele', createIndex=['FullName', 'Username'])`

set(**args*, ***kwargs*)
Refer to: *set()*

skip(*skip*)
Sets the number of documents to skip in the result

Parameters **skip** – the number of documents to skip

sort(**sort_tuples*)
pymongo-style sorting. Accepts a list of tuples.

Parameters **sort_tuples** – varargs of sort tuples.

unset(*qfield*)
Refer to: *unset()*

class ommongo.query.**QueryResult**(*session*, *cursor*, *type*, *raw_output=False*, *fields=None*)

clone()

next()

rewind()

Mongo Query Expression Language

Query Fields

class ommongo.query_expression.**QueryField**(*type*, *parent=None*)

elem_match(*value*)
This method does two things depending on the context:

1. In the context of a query expression it:

Creates a query expression to do an $elemMatch on the selected field. If the type of this field is a DocumentField the value can be either a QueryExpression using that Document's fields OR you can use a dict for raw mongo.

See the mongo documentation for thorough treatment of elemMatch: http://docs.mongodb.org/manual/reference/operator/elemMatch/

2. In the context of choosing fields in a query.fields() expr:

Sets the field to use elemMatch, so only the matching elements of a list are used. See the mongo docs for more details: http://docs.mongodb.org/manual/reference/projection/elemMatch/

endswith(*suffix*, *ignore_case=False*, *options=None*)
A query to check if a field ends with a given suffix string

Example: `session.query(Spell).filter(Spells.name.endswith("cadabra", ignore_case=True))`

eq_(*value*)
Creates a query expression where `this field == value`

Note: The prefered usage is via an operator: `User.name == value`

exclude()
Use in a query.fields() expression to say this field should be excluded. The default of fields() is to include only fields which are specified. This allows retrieving of "every field except 'foo'".

exists(*exists=True*)
Create a MongoDB query to check if a field exists on a Document.

fields_expression

ge_(*value*)
Creates a query expression where `this field >= value`

Note: The prefered usage is via an operator: `User.name >= value`

get_absolute_name()
Returns the full dotted name of this field

get_type()
Returns the underlying *ommongo.fields.Field*

gt_(*value*)
Creates a query expression where `this field > value`

Note: The prefered usage is via an operator: `User.name > value`

in_(**values*)
A query to check if this query field is one of the values in `values`. Produces a MongoDB `$in` expression.

le_(*value*)
Creates a query expression where `this field <= value`

Note: The prefered usage is via an operator: `User.name <= value`

lt_(*value*)
Creates a query expression where `this field < value`

Note: The prefered usage is via an operator: `User.name < value`

matched_index()
Represents the matched array index on a query with objects inside of a list. In the MongoDB docs, this is the `$` operator

ne_ (*value*)
Creates a query expression where `this field != value`

> **Note:** The prefered usage is via an operator: `User.name != value`

near (*x*, *y*, *max_distance=None*)
Return documents near the given point

near_sphere (*x*, *y*, *max_distance=None*)
Return documents near the given point using sphere distances

nin (**values*)
A query to check if this query field is not one of the values in `values`. Produces a MongoDB `$nin` expression.

regex (*expression*, *ignore_case=False*, *options=None*)
A query to check if a field matches a given regular expression :param ignore_case: Whether or not to ignore the case (setting this to True is the same as setting the 'i' option) :param options: A string of option characters, as per the MongoDB $regex operator (e.g. "imxs")

Example: `session.query(Spell).filter(Spells.name.regex(r'^abra[a-z]*cadabra$', ignore_case=True))`

startswith (*prefix*, *ignore_case=False*, *options=None*)
A query to check if a field starts with a given prefix string

Example: `session.query(Spell).filter(Spells.name.startswith("abra", ignore_case=True))`

> **Note:** This is a shortcut to .regex('^' + re.escape(prefix)) MongoDB optimises such prefix expressions to use indexes appropriately. As the prefix contains no further regex, this will be optimized by matching only against the prefix.

within_box (*corner1*, *corner2*)
Adapted from the Mongo docs:

```
session.query(Places).filter(Places.loc.within_box(cornerA, cornerB)
```

within_polygon (*polygon*)
Adapted from the Mongo docs:

```
polygonA = [ [ 10, 20 ], [ 10, 40 ], [ 30, 40 ], [ 30, 20 ] ]
polygonB = { a : { x : 10, y : 20 }, b : { x : 15, y : 25 }, c : { x : 20, y␣
↪: 20 } }
session.query(Places).filter(Places.loc.within_polygon(polygonA)
session.query(Places).filter(Places.loc.within_polygon(polygonB)
```

within_radius (*x*, *y*, *radius*)
Adapted from the Mongo docs:

```
session.query(Places).filter(Places.loc.within_radius(1, 2, 50)
```

within_radius_sphere (*x*, *y*, *radius*)
Adapted from the Mongo docs:

```
session.query(Places).filter(Places.loc.within_radius_sphere(1, 2, 50)
```

Query Expressions

class ommongo.query_expression.**QueryExpression**(*obj*)

A QueryExpression wraps a dictionary representing a query to perform on a mongo collection. The

---

Note: There is no `and_` expression because multiple expressions can be specified to a single call of `Query.filter()`

---

not_()

Negates this instance's query expression using MongoDB's `$not` operator

Example: `(User.name == 'Jeff').not_()`

Note: Another usage is via an operator, but parens are needed to get past precedence issues: `~ (User.name == 'Jeff')`

---

or_(*expression*)

Adds the given expression to this instance's MongoDB `$or` expression, starting a new one if one does not exst

Example: `(User.name == 'Jeff').or_(User.name == 'Jack')`

Note: The prefered usageis via an operator: `User.name == 'Jeff' | User.name == 'Jack'`

Update Expressions

Updates are done by calling any of the update operations on a query object

class ommongo.update_expression.**UpdateExpression**(*query*)

add_to_set(*qfield*, *value*)

Atomically add `value` to `qfield`. The field represented by `qfield` must be a set

---

Note: Requires server version **1.3.0+**.

---

append(*qfield*, *value*)

Atomically append `value` to `qfield`. The operation will if the field is not a list field

execute()

Execute the update expression on the database

extend(*qfield*, **value*)

Atomically append each value in `value` to the field `qfield`

inc(**args*, ***kwargs*)

Atomically increment `qfield` by `value`

multi()
Update multiple documents. The Mongo default is to only update the first matching document

pop_first(*qfield*)
Atomically pop the first item in `qfield.` .. note:: Requires version **1.1+**

pop_last(*qfield*)
Atomically pop the last item in `qfield.` .. note:: Requires version **1.1+**

remove(*qfield*, *value*)
Atomically remove `value` from `qfield`

remove_all(*qfield*, **value*)
Atomically remove each value in `value` from `qfield`

safe(*safe=True*)
Mark the query as a "safe" query with pymongo.

Parameters **safe** – Defaults to True. Force "safe" on or off

set(**args*, ***kwargs*)
Usage is either:

set(self, qfield, value): Atomically set `qfield` to `value`

OR

set(key1=value1, key2=value2): Atomically set the named arguments on the current object to the values given. This form cannot update a sub-document

unset(*qfield*)
Atomically delete the field `qfield` .. note:: Requires server version **>= 1.3.0+**.

upsert()
If a document matching the query doesn't exist, create one

Remove Queries

Remove queries are executed with the `remove_query` method on a session. They have the same filtering methods as querying.

class `ommongo.query.`**RemoveQuery**(*type*, *session*)
Execute a remove query to remove the matched objects from the database

Parameters

- **type** – A subclass of class:*ommongo.document.Document*
- **db** – The *Session* which this query is associated with.

execute()
Run the remove command on the session. Return the result of `getLastError` if `safe` is `True`

filter(**query_expressions*)
Filter the remove expression with `*query_expressions`, as in the `Query` filter method.

filter_by(***filters*)
Filter for the names in `filters` being equal to the associated values. Cannot be used for sub-objects since keys must be strings

in_(*qfield*, **values*)
Works the same as the query expression method `in_`

nin(*qfield*, **values*)
Works the same as the query expression method `nin_`

or_(*first_qe*, **qes*)
Works the same as the query expression method `or_`

query

set_safe(*is_safe*, ***kwargs*)
Set this remove to be safe. It will call getLastError after the remove to make sure it was successful. `**kwargs` are parameters to MongoDB's getLastError command (as in pymongo's remove).

Find and Modify

Find and modify is done by calling find_and_modify on a query object

class ommongo.update_expression.**FindAndModifyExpression**(*query*, *new*, *remove*)

add_to_set(*qfield*, *value*)
Atomically add `value` to `qfield`. The field represented by `qfield` must be a set

---

Note: Requires server version **1.3.0+**.

---

append(*qfield*, *value*)
Atomically append `value` to `qfield`. The operation will if the field is not a list field

execute()
Execute the find and modify expression on the database

extend(*qfield*, **value*)
Atomically append each value in `value` to the field `qfield`

inc(**args*, ***kwargs*)
Atomically increment `qfield` by `value`

multi()
Update multiple documents. The Mongo default is to only update the first matching document

pop_first(*qfield*)
Atomically pop the first item in `qfield.` .. note:: Requires version **1.1+**

pop_last(*qfield*)
Atomically pop the last item in `qfield.` .. note:: Requires version **1.1+**

remove(*qfield*, *value*)
Atomically remove `value` from `qfield`

remove_all(*qfield*, **value*)
Atomically remove each value in `value` from `qfield`

safe(*safe=True*)
Mark the query as a "safe" query with pymongo.

Parameters **safe** – Defaults to True. Force "safe" on or off

set(**args*, ***kwargs*)
Usage is either:

set(self, qfield, value): Atomically set `qfield` to `value`

OR

set(key1=value1, key2=value2): Atomically set the named arguments on the current object to the values given. This form cannot update a sub-document

unset(*qfield*)
Atomically delete the field `qfield` .. note:: Requires server version **>= 1.3.0+**.

upsert()
If a document matching the query doesn't exist, create one

1.3.4 Exceptions

Field Exceptions

exception ommongo.exceptions.**BadValueException**(*name*, *value*, *reason*, *cause=None*)
An exception which is raised when there is something wrong with a value

exception ommongo.exceptions.**BadFieldSpecification**
An exception that is raised when there is an error in creating a field

Document Exceptions

exception ommongo.exceptions.**DocumentException**
Base for all document-related exceptions

exception ommongo.exceptions.**MissingValueException**
Raised when a required field isn't set

exception ommongo.exceptions.**ExtraValueException**
Raised when a value is passed in with no corresponding field

exception ommongo.exceptions.**FieldNotRetrieved**
If a partial document is loaded from the database and a field which wasn't retrieved is accessed this exception is raised

exception ommongo.exceptions.**InvalidConfigException**
Raised when a bad value is passed in for a configuration that expects its values to obey certain constraints.

Data Exceptions

exception ommongo.exceptions.**BadReferenceException**

exception ommongo.exceptions.**BadValueException**(*name*, *value*, *reason*, *cause=None*)
An exception which is raised when there is something wrong with a value

Session Exceptions

exception ommongo.exceptions.**TransactionException**
Exception which occurs when an invalid operation is called during a transaction

exception ommongo.exceptions.**SessionCacheException**
Exception when an error has occured with the MA caching mechanism

1.4 Examples

No examples yet. There are however code snippets throughout the documentation and a few files here: https://github.com/bapakode/OmMongo/tree/master/examples

CHAPTER 2

Introduction

OmMongo is a layer on top of the *Python MongoDB* driver which adds client-side schema definitions, an easier to work with and programmatic query language, and a Document-Object mapper which allows python objects to be saved and loaded into the database in a type-safe way.

An explicit goal of this project is to be able to perform as many operations as possible without having to perform a load/save cycle since doing so is both significantly slower and more likely to cause data loss.

There's more detail in the tutorial, but a small example is on this page below the contents of the documentation.

A Tour of OmMongo's Features

CHAPTER 3

Object-Document Mapping

Turn MongoDB documents into Python objects and vice-versa, add schemas and validatation to your models, and allow the separation of the python and mongo representations of your data.

- Support for all of the basic mongo types.
- Separate Python and Mongo names. Field and collection names can be overridden.
- Indexing on dict keys. MA provides a dict-like field which can have arbitrary key and value types as well as allowing indexing on the keys — not normally possible in Mongo. See: *KVField*
- Computed values, generate from other fields. See: *ComputedField*
- Created and Modified fields based on the computed fields which record the date something was first created or last updated.
- Timezone support. A timezone can be passed using pytz and all dates will have timezone data attached and be converted to the given timezone.
- User-defined validation — Provide your own validation functions for simple validations of fields. See *Field*
- User-defined fields — For more customization, entirely new fields can be created
- A field that can hold arbitrary values: *AnythingField*
- Validation happens at assignment time, so you'll know exactly where
- Indexes are defined on the class

CHAPTER 4

Sessions and Query Language

- Type-safe queries. Queries are validated to make sure the values passed are allowed in the query fields.
- Faux-Transactions. When using the session with the `with` statement updates are accumulated until the block is done, making it much less likely that a python error will leave your database in a bad state.
- Automatically Calculated Updates — The session object has an *ommongo.session.Session.update()* function which determines which fields are dirty and will execute the appropriate update operations to update the object in the database.
- Drop into raw Mongo. Most functions will accept raw pymongo instead of the ommongo objects. Type safety will be maintained either way

  For example:

```
session.query('SomeClass').filter(SomeClass.name == foo).limit(5)
```

  is perfectly valid, as is:

```
session.query(SomeClass).filter({'name':'foo'})
```

CHAPTER 5

Installation

```
pip install OmMongo
```

You can also download the source code from the Python Package index or GitHub:

The source code is available at: https://github.com/bapakode/OmMongo

The PyPi page is located here: https://pypi.python.org/pypi/OmMongo/

CHAPTER 6

Examples

```
>>> from ommongo.session import Session
>>> from ommongo.document import Document, Index
>>> from ommongo.fields import *
>>> # Subclasses of Document both provide the mapping needed for
... # queries as well as the classes for loading/saving objects.
... class User(Document):
...     config_collection_name = 'users'
...
...     # Setting the possible values by using fields
...     first_name = StringField()
...     last_name = StringField()
...     age = IntField(min_value=0, required=False)
...
...     # db_field allows a different DB field name than the one on the
...     # python object
...     email = StringField(db_field='email_address')
...     bio = StringField(max_length=1000, required=False)
...
...     # A computed field decorator allows values
...     @computed_field(SetField(StringField()), deps=[bio])
...     def keywords(obj):
...         return set(obj.get('bio','').split(' '))
...
...     kw_index = Index().ascending('keywords')
...     name_index = Index().descending('first_name').ascending('last_name')
...     email_index = Index().descending('email').unique()
...
...     def __eq__(self, other):
...         return self.email == other.email
...
...     def __repr__(self):
...         return 'User(email="%s")' % self.email
...
>>> me = User(first_name='Bapak', last_name='Kode', email='opensource@bapakode.org',
```

(continues on next page)

(continued from previous page)

```
...     bio='Bapakode is the author of OmMongo')
>>>
>>> me.keywords
set(['author', 'of', 'is', 'Bapakode', 'OmMongo', 'the'])
>>>
>>> # This connections to the DB and starts the session
... session = Session.connect('ommongo-intro')
>>> session.clear_collection(User) # clear previous runs of this code!
>>>
>>> # Insert on a session will infer the correct collection and push the object
... # into the database
... session.save(me)
>>> set(['author', 'of', 'is', 'Bapakode', 'OmMongo', 'the'])
>>>
>>> # Get a user with me's email address and OmMongo in their bio (via keywords)
... db_user = session.query(User).filter(User.email == 'opensource@bapakode.org').in_
↪(User.keywords, 'OmMongo').one()
>>>
>>> db_user == me
True
>>>
>>> # Using filter_by for simple equality checking is easier
... session.query(User).filter_by(email='opensource@bapakode.org').in_(User.keywords,
↪'OmMongo').one()
User(email="opensource@bapakode.org")
>>>
>>> # It's also possible to do raw mongo filtering
... session.query(User).filter({'email':'opensource@bapakode.org', 'keywords':{'$in':[
↪'OmMongo']}}).one()
User(email="opensource@bapakode.org")
>>>
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

E

F

G

H

I