MongoKit Documentation

Release 1.0.0

Nicolas Clairon

February 08, 2017

Contents

MongoKit is a python module that brings structured schema and validation layer on top of the great pymongo driver. It has be written to be simpler and lighter as possible with the KISS and DRY principles in mind.

CHAPTER 1

Philosophy

MongoKit is designed to be:

- **Simple**: MongoKit use plain python type to describe document structure
- **Fast**: MongoKit is fast but if you *really* need to be fast you have access to the raw pymongo layer without changing the API
- **Powerful**: MongoKit brings many feature like document auto-reference, custom types or i18n support.

Your data is clean:

"Tools change, not data". In order to follow this "credo", MongoKit won't add any information into your data saved into the database. So if you need to use other mongo tools or ODMs in other languages, your data won't be polluted by MongoKit's stuff.

CHAPTER 2

Features

- Schema validation (which uses simple python types for the declaration)
- Schema-less feature
- Dot notation
- Nested and complex schema declaration
- Untyped field support
- Required fields validation
- Default values
- Custom validators
- Cross database document reference
- Random query support (which returns a random document from the database)
- Inheritance and polymorphism support
- Versionized document support (in beta stage)
- Partial auth support (it brings a simple User model)
- Operator for validation (currently : OR, NOT and IS)
- Simple web framework integration
- Import/export to json
- I18n support
- GridFS support
- Document migration support

Quick Example

A quick example

Documents are enhanced python dictionaries with a *validate()* method. A Document declaration look as follows:

```
>>> # Python 3
>>> from mongokit import *
>>> import datetime

>>> connection = Connection()

>>> @connection.register
... class BlogPost(Document):
...     structure = {
...             'title':str,
...             'body':str,
...             'author':str,
...             'date_creation':datetime.datetime,
...             'rank':int
...     }
...     required_fields = ['title','author', 'date_creation']
...     default_values = {'rank':0, 'date_creation':datetime.datetime.utcnow}

>>> # Python 2
>>> from mongokit import *
>>> import datetime

>>> connection = Connection()

>>> @connection.register
... class BlogPost(Document):
...     structure = {
...             'title':unicode,
...             'body':unicode,
...             'author':unicode,
...             'date_creation':datetime.datetime,
...             'rank':int
...     }
...     required_fields = ['title','author', 'date_creation']
...     default_values = {'rank':0, 'date_creation':datetime.datetime.utcnow}
```

We establish a connection and register our objects.

```
>>> blogpost = con.test.example.BlogPost() # this uses the database "test" and the collection "examp
>>> blogpost['title'] = 'my title'
>>> blogpost['body'] = 'a body'
>>> blogpost['author'] = 'me'
>>> blogpost
{'body': 'a body', 'title': 'my title', 'date_creation': datetime.datetime(...), 'rank': 0, 'author'
>>> blogpost.save()
```

Saving the object will call the *validate()* method.

And you can use a more complex structure as follows:

```
>>>  @connection.register
...  class ComplexDoc(Document):
...     __database__ = 'test'
...     __collection__ = 'example'
...     structure = {
...         "foo" : {"content":int},
...         "bar" : {
...             'bla':{'spam':int}
...         }
...     }
...     required_fields = ['foo.content', 'bar.bla.spam']
```

CHAPTER 4

Community

Suggestions and patches are really welcome. If you find mistakes in the documentation feel free to send a pull request or to contact us.

- Google Groups
- Github Issues
- Stackoverflow

CHAPTER 5

Contents:

5.1 Getting Started

MongoKit is based on pymongo. As such, all of the pymongo API is exposed through MongoKit. If you don't find what you want in the MongoKit API, please take a look at pymongo's documentation. All the pymongo API is exposed via connection, database and collection so Connection, Database and Collection are wrappers around pymongo objects.

In this example, we will create a simple MongoKit document definition and use it to create new instances of that document.

5.1.1 Defining a Document

Start by defining the Document:

```
# Python 3
from mongokit import Document, Connection
import datetime

class BlogPost(Document):
    structure = {
        'title': str,
        'body': str,
        'author': str,
        'date_creation': datetime.datetime,
        'rank': int,
        'tags': [str],
    }
    required_fields = ['title', 'author', 'date_creation']
    default_values = {
        'rank': 0,
        'date_creation': datetime.datetime.utcnow
    }


# Python 2
from mongokit import Document, Connection
import datetime

class BlogPost(Document):
    structure = {
        'title': basestring,
        'body': basestring,
        'author': basestring,
        'date_creation': datetime.datetime,
        'rank': int,
        'tags': [basestring],
    }
    required_fields = ['title', 'author', 'date_creation']
    default_values = {
        'rank': 0,
        'date_creation': datetime.datetime.utcnow
    }
```

The `structure` is simply a dictionary with python type. In this example, `title` must be a string and `rank` must be an int. For more information, see `structure`.

Optionally, you can add some descriptors. In order to specify fields which are required, just add a `required_fields` attribute. This is a simple list which lists all required_fields (ie, this field must not be None when validating). You can also specify the `default_values` attribute which can indicates the default values of a Documents structure. Note that you can pass callable object (like a `datetime.utcnow`). For more information, see `descriptors`.

5.1.2 Connecting to MongoDB

Now, to open a connection to MongoDB:

```
connection = Connection()
```

This a wrapped version of pymongo's Connection and will attempt to connect to a MongoDB instance running locally.

e.g. Speficying a host

```
connection = Connection(host="HOSTNAME", port=PORT)
```

e.g. Specifying a Replica Set host

```
from mongokit import ReplicaSetConnection

connection = ReplicaSetConnection(
    host="HOSTNAME:PORT,HOSTNAME:PORT"
    replicaset="REPLICA_SET_NAME",
    read_preferences=pymongo.read_preferences.ReadPreferance.SECONDARY_PREFERRED)
```

For more information on how to configure the Connection, see pymongo's documentation.

Once you have an active connection, you need to register your `BlogPost` object:

```
connection.register([BlogPost])
```

Alternatively, the register method can be used as a decorator:

```
@connection.register
class BlogPost(Document):
    structure = {...}
```

Now, let's create a new blogpost in the "blog_posts" collection in the database "blog". In pymongo's syntax, you would use `connection.<database>.<collection>` to access the collection. Once you have the collection, you can create a new document:

```
>>> connection.test.blogpost.BlogPost()
{'body': None, 'title': None, 'author': None, 'rank': 0, 'date_creation': datetime.datetime(...), 'ta
```

Note that `date_creation` was automatically filled by `utcnow()` and rank is `0`.

To avoid repeating ourselves though, let's specify the database and collection name in the Document definition:

```
@connection.register
class BlogPost(Document):
    __collection__ = 'blog_posts'
    __database__ = 'blog'
    structure = {...}
```

Now, we can have access to our document directly from the connection:

```
>>> bp = connection.BlogPost()
>>> bp
{'body': None, 'title': None, 'author': None, 'rank': 0, 'date_creation': datetime.datetime(...), 'ta
```

5.1.3 Modifying the Document

Now let's modify our BlogPost and try the validation:

```
>>> bp['title'] = 1
>>> bp.validate()
Traceback (most recent call last):
...
SchemaTypeError: title must be an instance of basestring not int
```

Alright, type validation works. :

```
>>> bp['title'] = 'my first blog post'
```

`validate` method will also check if all the required fields are set:

```
>>> bp.validate()
Traceback (most recent call last):
...
RequireFieldError: author is required

>>> bp['author'] = 'myself'
>>> bp.validate()
>>>
```

Now let's save our new blogpost to the database:

```
>>> bp.save()
```

Note that `save` will call the `validate` method, so you don't have to validate each time.

5.1.4 Querying the Database

Once you've got data in the database, you can quickly retrieve your blog posts as well:

```
>>> for post in connection.BlogPost.find():
...     print post['title']
...
my first blog post
```

pymongo makes it very easy to perform complex queries on your data so for more information, see the CRUD Operations documentation.

5.2 Mapper Syntax

5.2.1 Documents

Documents are the basic building blocks of MongoKit. They define the schema of each document and how that document should be accessed

Document Class

```python
from mongokit import Document, Connection

class MyDocument(Document):
    structure = {
        ...
    }
    required_fields = [
        ...
    ]
    default_values = {
        ...
    }
```

You can read more about the structure attribute, and the `required_fields` and `default_values` descriptors. They are the primary definition of a document. MongoKit also supports handling i18n, indexes, and migration.

Registering

Once a document has been defined, it must be registered with a Connection:

```python
connection = Connection()
connection.register([MyDocument])
```

Optionally, the register method can be used as a decorator:

```python
@connection.register
class MyDocument(Document):
    structure = {...}
```

Database and Collection

To use a Document, you must call it from a collection. In pymongo's syntax, you would use `connection.<database>.<collection>` to access the collection. Once you have the collection, you can create a new document:

```python
>>> connection.database.collection.MyDocument()
{... new Document's default values ...}
```

As a short cut, you can define the database and collection names in the Document definition:

```
@connection.register
class MyDocument(Document):
    __collection__ = 'collection_name'
    __database__ = 'database_name'
    structure = {...}
```

Now, we can have access to our document directly from the connection:

```
>>> connection.MyDocument()
{... new Document's default values ...}
```

Note that if you want to specify the `__database__`, you should also specify the `__collection__` attribute.

It is also possible to access the Document from the database:

```
>>> connection.database.MyDocument() # this will use __collection__ as collection name
```

This matches the typical pattern of creating and passing around a `db` object:

```
>>> connection = Connection()
>>> db = connection[MONGODB_DATABASE_NAME]
>>> db.MyDocument()
```

Changing Collection Dynamically

You might need to specify a different db or collection dynamically. For instance, say you want to store a User by database.

```
>>> # Python 3
>>> class User(Document):
...     structure = {
...         'login':str,
...         'screen_name':str
...     }
>>> con.register([User])
```

```
>>> # Python 2
>>> class User(Document):
...     structure = {
...         'login':unicode,
...         'screen_name':unicode
...     }
>>> con.register([User])
```

Like pymongo, MongoKit allow you to change those parameters on the fly :

```
>>> user_name = 'namlook'
>>> user_collection = connection[user_name].profile
returns a reference to the database 'namlook' in the collection 'profile'.
```

Now, we can query the database by passing our new collection :

```
>>> profiles = user_collection.User.find()
```

```
>>> user = user_collection.User()
>>> user['login'] = 'namlook@namlook.com'
>>> user['screen_name'] = 'Namlook'
```

Calling `user.save()` will save the object into the database `namlook` in the collection `profile`.

Dot Notation

If you want to use the dot notation (ala json), you must set the `use_dot_notation` attribute to True:

```
# Python 3
class TestDotNotation(Document):
    use_dot_notation = True

    structure = {
        'foo':{
            'bar': str
        }
    }

# Python 2
class TestDotNotation(Document):
    use_dot_notation = True

    structure = {
        'foo':{
            'bar': basestring
        }
    }
```

```
>>> connection.register([TestDotNotation])
>>> doc = connection.database.TestDotNotation()
>>> doc.foo.bar = 'blah'
>>> doc
{'foo': {'bar': 'blah'}}
```

Note that if an attribute is not in structure, the value will be added as attribute :

```
>>> doc.arf = 3 # arf is not in structure
>>> doc
{'foo': {'bar': u'bla'}}
```

If you want to be warned when a value is set as attribute, you can set the *dot_notation_warning* attribute as True.

Polymorphism

In the following example, we have two objects, A and B, which inherit from Root. And we want to build an object C from A and B. Let's build Root, A and B first:

```
# Python 3
from mongokit import *
class Root(Document):
    structure = {
        'root': int
    }
    required_fields = ['root']

class A(Root):
    structure = {
        'a_field': str,
    }
    required_fields = ['a_field']
```

```
class B(Root):
    structure = {
        'b_field': str,
    }

# Python 2
from mongokit import *
class Root(Document):
    structure = {
        'root': int
    }
    required_fields = ['root']

class A(Root):
    structure = {
        'a_field': basestring,
    }
    required_fields = ['a_field']


class B(Root):
    structure = {
        'b_field': basestring,
    }
```

Polymorphisms just work as expected:

```
class C(A,B):
    structure = {'c_field': float}

>>> c = C()
>>> c == {'b_field': None, 'root': None, 'c_field': None, 'a_field': None}
True
>>> C.required_fields
['root', 'a_field']
```

5.2.2 Descriptors

In the MongoKit philosophy, the structure must be simple, clear and readable. So all descriptors (validation, requirement, default values, etc.) are described outside the structure. Descriptors can be combined and applied to the same field.

required

This descriptor describes the required fields:

```
# Python 3
class MyDoc(Document):
    structure = {
        'bar': str,
        'foo':{
            'spam': str,
            'eggs': int,
        }
    }
    required = ['bar', 'foo.spam']
```

```
# Python 2
class MyDoc(Document):
    structure = {
        'bar': basestring,
        'foo':{
            'spam': basestring,
            'eggs': int,
        }
    }
    required = ['bar', 'foo.spam']
```

If you want to reach nested fields, just use the dot notation.

default_values

This descriptor allows to specify a default value at the creation of the document:

```
# Python 3
class MyDoc(Document):
     structure = {
         'bar': str,
         'foo':{
             'spam': str,
             'eggs': int,
         }
     }
     default_values = {'bar': 'hello', 'foo.eggs': 4}

# Python 2
class MyDoc(Document):
     structure = {
         'bar': basestring,
         'foo':{
             'spam': basestring,
             'eggs': int,
         }
     }
     default_values = {'bar': 'hello', 'foo.eggs': 4}
```

Note that the default value must be a valid type. Again, to reach nested fields, use dot notation.

validators

This descriptor brings a validation layer to a field. It takes a function which returns `False` if the validation fails, `True` otherwise:

```
# Python 3
import re
def email_validator(value):
   email = re.compile(r'(?:^|\s)[-a-z0-9_.]+@(?:[-a-z0-9]+\.)+[a-z]{2,6}(?:\s|$)',re.IGNORECASE)
   return bool(email.match(value))

class MyDoc(Document):
   structure = {
      'email': str,
      'foo': {
            'eggs': int,
        }
    }
    validators = {
        'email': email_validator,
        'foo.eggs': lambda x: x > 10
    }

# Python 2
import re
def email_validator(value):
    email = re.compile(r'(?:^|\s)[-a-z0-9_.]+@(?:[-a-z0-9]+\.)+[a-z]{2,6}(?:\s|$)',re.IGNORECASE)
    return bool(email.match(value))

class MyDoc(Document):
    structure = {
        'email': basestring,
        'foo': {
            'eggs': int,
        }
    }
    validators = {
        'email': email_validator,
        'foo.eggs': lambda x: x > 10
    }
```

You can add custom message to your validators by throwing a `ValidationError` instead of returning `False`

```
def email_validator(value):
    email = re.compile(r'(?:^|\s)[-a-z0-9_.]+@(?:[-a-z0-9]+\.)+[a-z]{2,6}(?:\s|$)',re.IGNORECASE)
    if not email.match(value):
        raise ValidationError('%s is not a valid email' % value)
```

Make sure to include one '%s' in the message. This will be used to refer to the name of the field containing errors.

You can also pass params to your validator by wrapping it in a class:

```
class MinLengthValidator(object):
    def __init__(self, min_length):
        self.min_length = min_length

    def __call__(self, value):
        if len(value) >= self.min_length:
            return True
        else:
            raise Exception('%s must be at least %d characters long.' % (value, self.min_length))

# Python 3
class Client(Document):
    structure = {
      'first_name': str
    }
    validators = { 'first_name': MinLengthValidator(2) }

# Python 2
class Client(Document):
    structure = {
      'first_name': basestring
    }
    validators = { 'first_name': MinLengthValidator(2) }
```

In this example, `first_name` must contain at least 2 characters.

Adding Complex Validation

If the use of a validator is not enough, you can overload the validation method to fit your needs.

For example, take the following document:

```
# Python 3
class MyDoc(Document):
    structure = {
        'foo': int,
        'bar': int,
        'baz': str
    }

# Python 2
class MyDoc(Document):
    structure = {
        'foo': int,
        'bar': int,
        'baz': basestring
    }
```

We want to be sure before saving our object that foo is greater than bar. To do that, we just overload the validation method:

```
def validate(self, *args, **kwargs):
    assert self['foo'] > self['bar']
    super(MyDoc, self).validate(*args, **kwargs)
```

Skipping Validation

Once your application is ready for production and you are sure that the data is consistent, you might want to skip the validation layer. This will make MongoKit significantly faster (as fast as pymongo). In order to do that, just set the `skip_validation` attribute to `True`.

TIP: It is a good idea to create a `RootDocument` and to inherit all your document classes from it. This will allow you to control the default behavior of all your objects by setting attributes on the RootDocument:

```
class RootDocument(Document):
    structure = {}
    skip_validation = True
    use_autorefs = True

class MyDoc(RootDocument):
    structure = {
        'foo': int
    }
```

Note that you can always force the validation at any moment on saving even if `skip_validation` is `True`:

```
>>> con.register([MyDoc]) # No need to register RootDocument as we do not instantiate it
>>> mydoc = tutorial.MyDoc()
>>> mydoc['foo'] = 'bar'
>>> mydoc.save(validate=True)
Traceback (most recent call last):
...
SchemaTypeError: foo must be an instance of int not basestring
```

Quiet Validation Detection

By default, when validation is on, each error raises an Exception. Sometimes, you just want to collect all errors in one place. This is possible by setting the `raise_validation_errors` to `False`. This causes all errors to be stored in the `validation_errors` attribute:

```
class MyDoc(Document):
    raise_validation_errors = False
    structure = {
        'foo': set,
    }
```

```
>>> con.register([MyDoc])
>>> doc = tutorial.MyDoc()
>>> doc.validate()
>>> doc.validation_errors
{'foo': [StructureError("<type 'set'> is not an authorized type",), RequireFieldError('foo is require
```

`validation_errors` is a dictionary which takes the field name as key and the Python exception as value. There are two issues with foo here: one with it's structure (`set` is not an authorized type) and another with required field (`foo` is required field but is not specified).

```
>>> doc.validation_errors['foo'][0].message
"<type 'set'> is not an authorized type"
```

Validate Keys

If the value of key is not known but we want to validate some deeper structure, we use the "$<type>" descriptor:

```
# Python 3
class MyDoc(Document):
    structure = {
        'key': {
            str: {
                'first': int,
                'secondpart': {
                    str: int
                }
            }
        }
    }

required_fields = ["key1.$str.bla"]

# Python 2
class MyDoc(Document):
    structure = {
        'key': {
            unicode: {
                'first': int,
                'secondpart': {
                    unicode: int
                }
            }
        }
    }

required_fields = ["key1.$unicode.bla"]
```

Note that if you use a Python type as a key in structure, generate_skeleton won't be able to build the entire underlying structure :

```
>>> con.register([MyDoc])
>>> tutorial.MyDoc() == {'key1': {}, 'bla': None}
True
```

So, neither default_values nor validators will work.

5.2.3 The Structure

The `structure` is a simple dict which defines the document's schema.

Field Types

Field types are simple python types. By default, MongoKit allows the following types:

```
# Common types between python 3 and python 2

None # Untyped field
bool
int
float
list
dict
datetime.datetime
bson.binary.Binary
pymongo.objectid.ObjectId
bson.dbref.DBRef
bson.code.Code
type(re.compile(""))
uuid.UUID
CustomType

# Python 3 types
bytes
str

# Python 2 types
basestring
long
unicode
```

Untyped field

Sometimes you don't want to specify a type for a field. In order to allow a field to have any of the authorized types, just set the field to `None` in the structure:

```
class MyDoc(Document):
  structure = {
    'foo': int,
    'bar': None
  }
```

In this example, `bar` can be any of the above types except for a CustomType.

Nested Structure

MongoDB allows documents to include nested structures using lists and dicts. You can also use the structure dict to specify these nested structures as well.

Dicts

Python's dict syntax `{}` is used for describing nested structure:

```
# Python 3
class Person(Document):
  structure = {
    'biography': {
      'name': str,
      'age': int
    }
  }

# Python 2
class Person(Document):
  structure = {
    'biography': {
      'name': basestring,
      'age': int
    }
  }
```

This validates that each document has an author dict which contains a string name and an integer number of books.

If you want to nest a dict without type validation, you must use the dict type keyword instead:

```
class Person(Document):
  structure = {
    'biography': dict
  }
```

If you don't specify the nested structure or don't use the dict type keyword, you won't be able to add values to the nested structure:

```
class Person(Document):
  structure = {
    'biography': {}
  }
```

```
>>> bob = Person()
>>> bob['biography']['foo'] = 'bar'
>>> bob.validate()
Traceback (most recent call last):
...
StructureError: unknown fields : ['foo']
```

Using dict type is useful if you don't know what fields will be added *or* what types they will be. If you know the type of the field, it's better to explicitly specify it:

```
# Python 3
class Person(Document):
  structure = {
    'biography': {
      unicode: str
    }
  }

# Python 2
class Person(Document):
  structure = {
    'biography': {
      unicode: unicode
    }
  }
```

This will add another layer to validate the content. See the validate-keys section for more information.

Lists

The basic way to use a list is without validation of its contents:

```
class Article(Document):
    structure = {
        'tags': list
    }
```

In this example, the `tags` value must be a list but the contents of `tags` can be anything at all. To validate the contents of a list, you use Python's list syntax `[]` instead:

```
# Python 3
class Article(Document):
    structure = {
        'tags': [str]
    }

# Python 2
class Article(Document):
    structure = {
        'tags': [basestring]
    }
```

You can also validate an array of complex objects by using a dict:

```
# Python 3
class Article(Document):
    structure = {
        'tags': [
            {
            'name': str,
            'count': int
            }
        ]
    }

# Python 2
class Article(Document):
    structure = {
        'tags': [
            {
            'name': basestring,
            'count': int
            }
        ]
    }
```

Tuples

If you need a structured list with a fixed number of items, you can use tuple to describe it:

```
# Python 3
class MyDoc(Document):
    structure = {
        'book': (int, str, float)
    }

# Python 2
class MyDoc(Document):
    structure = {
        'book': (int, basestring, float)
    }
```

```
>>> con.register([MyDoc])
>>> mydoc = tutorial.MyDoc()
>>> mydoc['book']
[None, None, None]
```

Tuple are converted into a simple list and add another validation layer. Fields must follow the right type:

```
>>> # Python 3
>>> mydoc['book'] = ['foobar', 1, 1.0]
>>> mydoc.validate()
Traceback (most recent call last):
...
SchemaTypeError: book must be an instance of int not str

>>> # Python 2
>>> mydoc['book'] = ['foobar', 1, 1.0]
>>> mydoc.validate()
Traceback (most recent call last):
...
SchemaTypeError: book must be an instance of int not basestring
```

And they must have the right number of items:

```
>>> mydoc['book'] = [1, 'foobar']
>>> mydoc.validate()
Traceback (most recent call last):
...
SchemaTypeError: book must have 3 items not 2
```

As tuples are converted to list internally, you can make all list operations:

```
>>> mydoc['book'] = [1, 'foobar', 3.2]
>>> mydoc.validate()
>>> mydoc['book'][0] = 50
>>> mydoc.validate()
```

Sets

The `set` python type is not supported in pymongo. If you want to use it anyway, use the `Set()` custom type:

```
# Python 3
class MyDoc(Document):
  structure = {
    'tags': Set(str),
  }

# Python 2
class MyDoc(Document):
  structure = {
    'tags': Set(unicode),
  }
```

Using Custom Types

Sometimes we need to work with complex objects while keeping their footprint in the database fairly simple. Let's take a datetime object. A datetime object can be useful to compute complex date and though MongoDB can deal with datetime object, we may just want to store its unicode representation.

MongoKit allows you to work on a datetime object and store the unicode representation converted on the fly. In order to do this, we have to implement a CustomType and fill the custom_types attributes:

```
>>> import datetime
```

A CustomType object must implement two methods and one attribute:

- `to_bson(self, value)`: this method will convert the value to fit the correct authorized type before being saved in the db.
- `to_python(self, value)`: this method will convert the value taken from the db into a python object
- `validate(self, value, path)`: this method is optional and will add a validation layer. Please, see the *Set()* CustomType code for more example.
- You must specify a `mongo_type` property in the `CustomType` class. this will describe the type of the value stored in the mongodb.
- If you want more validation, you can specify a `python_type` property which is the python type the value will be converted to. It is a good thing to specify it as it make a good documentation.
- `init_type` attribute will allow to describe an empty value. For example, if you implement the python set as CustomType, you'll set `init_type` to `Set`. Note that `init_type` must be a type or a callable instance.

```
# Python 3
class CustomDate(CustomType):
    mongo_type = str
    python_type = datetime.datetime # optional, just for more validation
    init_type = None # optional, fill the first empty value

    def to_bson(self, value):
        """convert type to a mongodb type"""
        return unicode(datetime.datetime.strftime(value,'%y-%m-%d'))

    def to_python(self, value):
        """convert type to a python object"""
        if value is not None:
           return datetime.datetime.strptime(value, '%y-%m-%d')

    def validate(self, value, path):
        """OPTIONAL : useful to add a validation layer"""
        if value is not None:
            pass # ... do something here

# Python 2
class CustomDate(CustomType):
    mongo_type = unicode
    python_type = datetime.datetime # optional, just for more validation
    init_type = None # optional, fill the first empty value

    def to_bson(self, value):
        """convert type to a mongodb type"""
        return unicode(datetime.datetime.strftime(value,'%y-%m-%d'))

    def to_python(self, value):
        """convert type to a python object"""
        if value is not None:
           return datetime.datetime.strptime(value, '%y-%m-%d')

    def validate(self, value, path):
        """OPTIONAL : useful to add a validation layer"""
        if value is not None:
            pass # ... do something here
```

Now, let's create a Document:

```
class Foo(Document):
    structure = {
        'foo':{
            'date': CustomDate(),
        },
    }
```

Now, we can create Foo objects and work with python datetime objects:

```
>>> foo = Foo()
>>> foo['_id'] = 1
>>> foo['foo']['date'] = datetime.datetime(2003,2,1)
>>> foo.save()
```

The object saved in the db has the unicode footprint as expected:

```
>>> tutorial.find_one({'_id':1})
{u'_id': 1, u'foo': {u'date': u'03-02-01'}}
```

Querying an object will automatically convert the CustomType into the correct python object:

```
>>> foo = tutorial.Foo.get_from_id(1)
>>> foo['foo']['date']
datetime.datetime(2003, 2, 1, 0, 0)
```

OR, NOT, and IS operators

You can also use boolean logic to do field type validation.

OR operator

Let's say that we have a field which can be either unicode, int or a float. We can use the OR operator to tell MongoKit to validate the field :

```
>>> # Python 3
>>> from mongokit import OR
>>> from datetime import datetime
>>> class Account(Document):
...     structure = {
...         "balance": {'foo': OR(str, int, float)}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['balance']['foo'] = '3.0'
>>> account.validate()
>>> account['balance']['foo'] = 3.0
>>> account.validate()
>>> # but
>>> account['balance']['foo'] = datetime.now()
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <str or int or float> not datetime
```

```
>>> # Python 2
>>> from mongokit import OR
>>> from datetime import datetime
>>> class Account(Document):
...     structure = {
...         "balance": {'foo': OR(unicode, int, float)}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['balance']['foo'] = u'3.0'
>>> account.validate()
>>> account['balance']['foo'] = 3.0
>>> account.validate()
>>> # but
>>> account['balance']['foo'] = datetime.now()
>>> account.validate()
```

```
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <unicode or int or float> not datetime
```

NOT operator

You can also use the NOT operator to tell MongoKit that you don't want a given type for a field :

```
>>> # Python 3
>>> from mongokit import NOT
>>> class Account(Document):
...     structure = {
...         "balance": {'foo': NOT(str, datetime)}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['balance']['foo'] = 3
>>> account.validate()
>>> account['balance']['foo'] = 3.0
>>> account.validate()
>>> # but
>>> account['balance']['foo'] = datetime.now()
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <not str, not datetime> not datetime
>>> account['balance']['foo'] = u'3.0'
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <not str, not datetime> not str
```

```
>>> # Python 2
>>> from mongokit import NOT
>>> class Account(Document):
...     structure = {
...         "balance": {'foo': NOT(unicode, datetime)}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['balance']['foo'] = 3
>>> account.validate()
>>> account['balance']['foo'] = 3.0
>>> account.validate()
>>> # but
>>> account['balance']['foo'] = datetime.now()
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <not unicode, not datetime> not datetime
>>> account['balance']['foo'] = u'3.0'
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: balance.foo must be an instance of <not unicode, not datetime> not unicode
```

IS operator

Sometimes you might want to define a field which accepts only values limited to a predefined set. The IS operator can be used for this purpose:

```
>>> # Python 3
>>> from mongokit import IS
>>> class Account(Document):
...     structure = {
...         "flag": {'foo': IS('spam', 'controversy', 'phishing')}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['flag']['foo'] = 'spam'
>>> account.validate()
>>> account['flag']['foo'] = 'phishing'
>>> account.validate()
>>> # but
>>> account['flag']['foo'] = 'foo'
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: flag.foo must be in ['spam', 'controversy', 'phishing'] not foo

>>> # Python 2
>>> from mongokit import IS
>>> class Account(Document):
...     structure = {
...         "flag": {'foo': IS(u'spam', u'controversy', u'phishing')}
...     }
>>> # Validation
>>> con.register([Account])
>>> account = tutorial.Account()
>>> account['flag']['foo'] = u'spam'
>>> account.validate()
>>> account['flag']['foo'] = u'phishing'
>>> account.validate()
>>> # but
>>> account['flag']['foo'] = u'foo'
>>> account.validate()
Traceback (most recent call last):
...
SchemaTypeError: flag.foo must be in [u'spam', u'controversy', u'phishing'] not foo
```

Schemaless Structure

One of the main advantages of MongoDB is the ability to insert schemaless documents into the database. As of version 0.7, MongoKit allows you to save partially structured documents. For now, this feature must be activated. It will be the default behavior in a future release.

To enable schemaless support, use the `use_schemaless` attribute:

```
class MyDoc(Document):
    use_schemaless = True
```

Setting `use_schemaless` to `True` allows to have an unset structure, however you can still specify a structure:

```
# Python 3
class MyDoc(Document):
    use_schemaless = True
    structure = {
        'title': str,
        'age': int
    }
    required_fields = ['title']

# Python 2
class MyDoc(Document):
    use_schemaless = True
    structure = {
        'title': basestring,
        'age': int
    }
    required_fields = ['title']
```

MongoKit will raise an exception only if required fields are missing:

```
>>> doc = MyDoc({'age': 21})
>>> doc.save()
Traceback (most recent call last):
...
StructureError: missed fields : ['title']
>>> doc = MyDoc({'age': 21, 'title': 'Hello World !'})
>>> doc.save()
```

5.2.4 Indexes

Sometimes, it's desirable to have indexes on your dataset - especially unique ones. In order to do that, you must fill the *indexes* attribute. The *indexes* attribute is a list of dictionary with the following structure:

"fields" # take a list of fields or a field name (required)

"unique" should this index guarantee uniqueness? (optional, False by default)

"ttl" (optional, 300 by default) time window (in seconds) during which this index will be recognized by subsequent calls to *ensure_index* - see pymongo documentation for *ensure_index* for details.

"check (optional, True by default) don't check if the field name is present in the structure. Useful if you don't know the field name.

Example:

```
>>> # Python 3
>>> class MyDoc(Document):
...     structure = {
...         'standard':str,
...         'other':{
...             'deep':str,
...         },
...         'notindexed':str,
...     }
...
...     indexes = [
...         {
...             'fields':['standard', 'other.deep'],
...             'unique':True,
...         },
...     ]

>>> # Python 2
>>> class MyDoc(Document):
...     structure = {
...         'standard':unicode,
...         'other':{
...             'deep':unicode,
...         },
...         'notindexed':unicode,
...     }
...
...     indexes = [
...         {
...             'fields':['standard', 'other.deep'],
...             'unique':True,
...         },
...     ]
```

or if you have more than one index:

```
>>> # Python 3
>>> class Movie(Document):
...     db_name = 'test'
...     collection_name = 'mongokit'
...     structure = {
...         'standard':str,
...         'other':{
...             'deep':str,
...         },
...         'alsoindexed':str,
...     }
...
...     indexes = [
...         {
...             'fields':'standard',
...             'unique':True,
...         },
...         {
...             'fields': ['alsoindexed', 'other.deep']
...         },
...     ]

>>> # Python 2
>>> class Movie(Document):
...     db_name = 'test'
...     collection_name = 'mongokit'
...     structure = {
...         'standard':unicode,
...         'other':{
...             'deep':unicode,
...         },
...         'alsoindexed':unicode,
...     }
...
...     indexes = [
...         {
...             'fields':'standard',
...                 'unique':True,
...             },
...             {
...                 'fields': ['alsoindexed', 'other.deep']
...             },
...         ]
```

By default, the index direction is set to 1. You can change the direction by passing a list of tuple. Direction must be one of *INDEX_ASCENDING* (or 1), *INDEX_DESCENDING* (or -1), *INDEX_OFF* (or 0), *INDEX_ALL* (or 2) or *INDEX_GEO2D* (or '2d'):

```
>>> # Python 3
>>> class MyDoc(Document):
...     structure = {
...         'standard':str,
...         'other':{
...             'deep':str,
...         },
...         'notindexed':str,
...     }
...
...     indexes = [
...         {
...             'fields':[('standard',INDEX_ASCENDING), ('other.deep',INDEX_DESCENDING)],
...             'unique':True,
...         },
...     ]

>>> # Python 2
>>> class MyDoc(Document):
...     structure = {
...         'standard':unicode,
...         'other':{
...             'deep':unicode,
...         },
...         'notindexed':unicode,
...     }
...
...     indexes = [
...         {
...             'fields':[('standard',INDEX_ASCENDING), ('other.deep',INDEX_DESCENDING)],
...             'unique':True,
...         },
...     ]
```

To prevent adding an index on the wrong field (misspelled for instance), MongoKit will check by default the *indexes* descriptor. In some cases may want to disable this. To do so, add `"check":True`:

```
>>> class MyDoc(Document):
...    structure = {
...        'foo': dict,
...        'bar': int
...    }
...    indexes = [
...        # I know this field is not in the document structure, don't check it
...        {'fields':['foo.title'], 'check':False}
...    ]
```

In this example, we index the field *foo.title* which is not explicitly specified in the structure.

5.2.5 Internationalization

Sometime you might want to present your data in differents languages and have i18n fields. Mongokit provides helper to do it.

i18n with dot_notation

Let's create a simple i18n BlogPost:

```
>>> # Python 3
>>> from mongokit import *
>>> class BlogPost(Document):
...     structure = {
...             'title':str,
...             'body':str,
...             'author':str,
...     }
...     i18n = ['title', 'body']
...     use_dot_notation = True
```

```
>>> # Python 2
>>> from mongokit import *
>>> class BlogPost(Document):
...     structure = {
...             'title':unicode,
...             'body':unicode,
...             'author':unicode,
...     }
...     i18n = ['title', 'body']
...     use_dot_notation = True
```

Declare your structure as usual and add an *i18n* descriptor. The *i18n* descriptor will tel Mongokit that the fields *title* and *body* will be in multiple language.

Note of the use of *use_dot_notation* attribute. Using i18n with dot notation is more fun but a little slower (not critical thought). We will see later how to use i18n is a blazing fast way (but less fun).

Let's create a BlogPost object and fill some fields:

```
>>> # Python 3
>>> con = Connection()
>>> con.register([BlogPost])
>>> blog_post = con.test.i18n.BlogPost()
>>> blog_post['_id'] = 'bp1'
>>> blog_post.title = "Hello"
>>> blog_post.body = "How are you ?"
>>> blog_post.author = "me"

>>> # Python 2
>>> con = Connection()
>>> con.register([BlogPost])
>>> blog_post = con.test.i18n.BlogPost()
>>> blog_post['_id'] = u'bp1'
>>> blog_post.title = u"Hello"
>>> blog_post.body = u"How are you ?"
>>> blog_post.author = u"me"
```

Now let's say we want to write your blog post in French. We select the language with the *set_lang()* method:

```
>>> # Python 3
>>> blog_post.set_lang('fr')
>>> blog_post.title = "Salut"
>>> blog_post.body = "Comment allez-vous ?"

>>> # Python 2
>>> blog_post.set_lang('fr')
>>> blog_post.title = u"Salut"
>>> blog_post.body = u"Comment allez-vous ?"
```

the *author* field is not i18n so we don't have to set it again.

Now let's play with our object

```
>>> # Python 3
>>> blog_post.title
'Salut'
>>> blog_post.set_lang('en')
>>> blog_post.title
'Hello'

>>> # Now, let's see how it work:
>>> blog_post
{'body': {'fr': 'Comment allez-vous ?', 'en': 'How are you ?'}, '_id': 'bp1', 'title': {'fr': 'Salut

>>> # Python 2
>>> blog_post.title
u'Salut'
>>> blog_post.set_lang('en')
>>> blog_post.title
u'Hello'

>>> # Now, let's see how it work:
>>> blog_post
{'body': {'fr': u'Comment allez-vous ?', 'en': u'How are you ?'}, '_id': u'bp1', 'title': {'fr': u'Sa
```

The *title* field is actually a dictionary which keys are the language and the values are the text. This is useful if you don't want to use the dot notation. Let's save our object:

```
>>> # Python 2
>>> blog_post.save()
>>> raw_blog_post = con.test.i18n.find_one({'_id':'bp1'})
>>> raw_blog_post
{'body': [{'lang': 'fr', 'value': 'Comment allez-vous ?'}, {'lang': 'en', 'value': 'How are you ?'}],

>>> # Python 3
>>> blog_post.save()
>>> raw_blog_post = con.test.i18n.find_one({'_id':'bp1'})
>>> raw_blog_post
{u'body': [{u'lang': u'fr', u'value': u'Comment allez-vous ?'}, {u'lang': u'en', u'value': u'How are
```

Now, the *title* field looks little different. This is a list of dictionary which have the following structure:

```
[{'lang': lang, 'value', text}, ...]
```

So, when an i18n object is save to the mongo database, it structure is changed. This is done to make indexation possible.

Note that you can still use this way event if you enable dot notation.

Default language

By default, the default language is english ('en'). You can change it easily by passing arguments in object creation:

```
>>> blog_post = con.test.i18n.BlogPost()
>>> blog_post.get_lang() # english by default
'en'
>>> blog_post = con.test.i18n.BlogPost(lang='fr')
>>> blog_post.get_lang()
'fr'
```

you can also specify a fallback language. This is useful if a field was translated yet:

```
>>> blog_post = con.test.i18n.BlogPost(lang='en', fallback_lang='en')
>>> blog_post.title = u"Hello"
>>> blog_post.set_lang('fr')
>>> blog_post.title # no title in french yet
u'Hello'
>>> blog_post.title = u'Salut'
>>> blog_post.title
u'Salut'
```

i18n without dot notation (the fast way)

If for you, speed is very very important, you might not want to use the dot notation (which brings some extra wrapping). While the API would be more fun, you can still use i18n. Let's take our BlogPost:

```
>>> # Python 3
>>> from mongokit import *
>>> class BlogPost(Document):
...     structure = {
...             'title':str,
...             'body':str,
...             'author':str,
...     }
...     i18n = ['title', 'body']

>>> con = Connection()
>>> con.register([BlogPost])
>>> blog_post = con.test.i18n.BlogPost()
>>> blog_post['_id'] = 'bp1'
>>> blog_post['title']['en'] = "Hello"
>>> blog_post['body']['en'] = "How are you ?"
>>> blog_post['author'] = "me"

>>> # Python 2
>>> from mongokit import *
>>> class BlogPost(Document):
...     structure = {
...             'title':unicode,
...             'body':unicode,
...             'author':unicode,
...     }
...     i18n = ['title', 'body']

>>> con = Connection()
>>> con.register([BlogPost])
>>> blog_post = con.test.i18n.BlogPost()
>>> blog_post['_id'] = u'bp1'
>>> blog_post['title']['en'] = u"Hello"
>>> blog_post['body']['en'] = u"How are you ?"
>>> blog_post['author'] = u"me"
```

As you can see, fields *title* and *body* are now dictionary which take the language as key. The result is the same:

```
>>> # Python 3
>>> blog_post
{'body': {'en': 'How are you ?'}, '_id': 'bp1', 'title': {'en': 'Hello'}, 'author': 'me'}

>>> # Python 2
>>> blog_post
{'body': {'en': u'How are you ?'}, '_id': u'bp1', 'title': {'en': u'Hello'}, 'author': u'me'}
```

The good thing is you don't have to use *set_lang()* and *get_lang()* anymore, the bad thing is you get some ugly:

```
>>> # Python 3
>>> blog_post['title']['fr'] = 'Salut'
>>> blog_post['title']
{'fr': 'Salut', 'en': 'Hello'}
```

```
>>> blog_post['body']['fr'] = 'Comment allez-vous ?'
>>> blog_post['body']
{'fr': 'Comment allez-vous ?', 'en': 'How are you ?'}
```

```
>>> # Python 2
>>> blog_post['title']['fr'] = u'Salut'
>>> blog_post['title']
{'fr': u'Salut', 'en': u'Hello'}
```

```
>>> blog_post['body']['fr'] = u'Comment allez-vous ?'
>>> blog_post['body']
{'fr': u'Comment allez-vous ?', 'en': u'How are you ?'}
```

Note that you don't have to fear to miss a i18n field. Validation will take care of that

```
>>> # Python 3
>>> blog_post['body'] = 'Comment allez-vous ?'
>>> blog_post.save()
Traceback (most recent call last):
...
SchemaTypeError: body must be an instance of i18n not unicode

>>> # Python 2
>>> blog_post['body'] = u'Comment allez-vous ?'
>>> blog_post.save()
Traceback (most recent call last):
...
SchemaTypeError: body must be an instance of i18n not unicode
```

i18n with different type

i18n in Mongokit was designed to handled any python types authorized in MongoKit. To illustrate, let's take a fake example : temperature.

```
>>> class Temperature(Document):
...     structure = {
...        "temperature":{
...           "degree": float
...        }
...     }
...     i18n = ['temperature.degree']
...     use_dot_notation = True

>>> con.register([Temperature])
>>> temp = con.test.i18n.Temperature()
>>> temp.set_lang('us')
>>> temp.temperature.degree = 75.2
>>> temp.set_lang('fr')
>>> temp.temperature.degree = 24.0
>>> temp.save()
```

This example describes that float can be translated too. Using i18n to handle temperature is a bad idea but you may find a useful usage of this feature.

Using i18n different type allow you to translate list:

```
>>> # Python 3
>>> class Doc(Document):
...     structure = {
...        "tags":[str]
...     }
...     i18n = ['tags']
...     use_dot_notation = True

>>> con.register([Doc])
>>> doc = con.test.i18n.Doc()
>>> doc.set_lang('en')
>>> doc.tags = ['apple', 'juice']
>>> doc.set_lang('fr')
>>> doc.tags = ['pomme', 'jus']
>>> doc
{'tags': {'fr': ['pomme', 'jus'], 'en': ['apple', 'juice']}}

>>> # Python 2
>>> class Doc(Document):
...     structure = {
...        "tags":[unicode]
...     }
...     i18n = ['tags']
...     use_dot_notation = True

>>> con.register([Doc])
>>> doc = con.test.i18n.Doc()
>>> doc.set_lang('en')
>>> doc.tags = [u'apple', u'juice']
>>> doc.set_lang('fr')
>>> doc.tags = [u'pomme', u'jus']
>>> doc
{'tags': {'fr': [u'pomme', u'jus'], 'en': [u'apple', u'juice']}}
```

5.2.6 Using DBRef

MongoKit has optional support for MongoDB's autoreferencing/dbref features. Autoreferencing allows you to embed MongoKit objects/instances inside another MongoKit object. With autoreferencing enabled, MongoKit and the pymongo driver will translate the embedded MongoKit object values into internal MongoDB DBRefs. The (de)serialization is handled automatically by the pymongo driver.

Autoreferences allow you to pass other Documents as values. Pymongo (with help from MongoKit) automatically translates these object values into DBRefs before persisting to Mongo. When fetching, it translates them back, so that you have the data values for your referenced object. See the autoref_sample. for further details/internals on this driver-level functionality. As for enabling it in your own MongoKit code, simply define the following class attribute upon your Document subclass:

```
use_autorefs = True
```

With autoref enabled, MongoKit's connection management will attach the appropriate BSON manipulators to your document's connection handles. We require you to explicitly enable autoref for two reasons:

- Using autoref and it's BSON manipulators (As well as DBRefs) can carry a performance penalty. We opt for performance and simplicity first, so you must explicitly enable autoreferencing.
- You may not wish to use auto-referencing in some cases where you're using DBRefs.

Once you have autoref enabled, MongoKit will allow you to define any valid subclass of Document as part of your document structure. **If your class does not define 'use_autorefs' as True, MongoKit's structure validation code will REJECT your structure.**

A detailed example

First let's create a simple doc:

```
>>> class DocA(Document):
...     structure = {
...         "a":{'foo':int},
...         "abis":{'bar':int},
...     }
...     default_values = {'a.foo':2}
...     required_fields = ['abis.bar']

>>> con.register([DocA])
>>> doca = tutorial.DocA()
>>> doca['_id'] = 'doca'
>>> doca['abis']['bar'] = 3
>>> doca.save()
```

Now, let's create a DocB which have a reference to DocA:

```
>>> class DocB(Document):
...     structure = {
...         "b":{"doc_a":DocA},
...     }
...     use_autorefs = True
```

Note that to be able to specify a Document into the structure, we must set *use_autorefs* as *True*.

```
>>> con.register([DocB])
>>> docb = tutorial.DocB()
```

The default value for an embedded doc is None:

```
>>> docb
{'b': {'doc_a': None}}
```

The validation acts as expected:

```
>>> docb['b']['doc_a'] = 4
>>> docb.validate()
Traceback (most recent call last):
...
SchemaTypeError: b.doc_a must be an instance of DocA not int

>>> docb['_id'] = 'docb'
>>> docb['b']['doc_a'] = doca
>>> docb
{'b': {'doc_a': {'a': {'foo': 2}, 'abis': {'bar': 3}, '_id': 'doca'}}, '_id': 'docb'}
```

Note that the reference can not only be cross collection but also cross database. So, it doesn't matter where you save the DocA object as long as it can be fetch with the same connection.

Now the interesting part. If we change a field in an embedded doc, the change will be done for all DocA which have the same '_id':

```
>>> docb['b']['doc_a']['a']['foo'] = 4
>>> docb.save()

>>> doca['a']['foo']
4
```

Required fields are also supported in embedded documents. Remember DocA have the 'abis.bar' field required. If we set it to None via the docb document, the RequireFieldError is raised:

```
>>> docb['b']['doc_a']['abis']['bar'] = None
>>> docb.validate()
Traceback (most recent call last):
...
RequireFieldError: abis.bar is required
```

About cross-database references

pymongo's DBRef doesn't take a database by default so MongoKit needs this information to fetch the correct Document.

An example is better than thousand words. Let's create an `EmbedDoc` and a `Doc` object:

```
>>> # Python 3
>>> class EmbedDoc(Document):
...    structure = {
...        "foo": str,
...    }

>>> class Doc(Document):
...    use_dot_notation=True
...    use_autorefs = True
...    structure = {
...        "embed": EmbedDoc,
...    }

>>> con.register([EmbedDoc, Doc])
>>> embed = tutorial.EmbedDoc()
>>> embed['foo'] = 'bar'
>>> embed.save()


>>> # Python 2
>>> class EmbedDoc(Document):
...     structure = {
...         "foo": unicode,
...     }

>>> class Doc(Document):
...     use_dot_notation=True
...     use_autorefs = True
...     structure = {
...         "embed": EmbedDoc,
...     }

>>> con.register([EmbedDoc, Doc])
>>> embed = tutorial.EmbedDoc()
>>> embed['foo'] = u'bar'
>>> embed.save()
```

Now let's insert a raw document with a DBRef but without specifying the database:

```
>>> raw_doc = {'embed':DBRef(collection='tutorial', id=embed['_id'])}
>>> doc_id = tutorial.insert(raw_doc)
```

Now what append when we want to load the data:

```
>>> doc = tutorial.Doc.get_from_id(doc_id)
Traceback (most recent call last):
...
RuntimeError: It appears that you try to use autorefs. I found a DBRef without database specified.
 If you do want to use the current database, you have to add the attribute `force_autorefs_current_d
 The DBRef without database is : DBRef(u'tutorial', ObjectId('4b6a949890bce72958000002'))
```

This mean that you may load data which could have been generated by map/reduce or raw data (like fixtures for instance) and the database information is not set into the DBRef. The error message tells you that you can add turn the *force_autorefs_current_db* as True to allow MongoKit to use the current collection by default (here 'test'):

```
>>> tutorial.database.name
u'test'
```

NOTE: You have to be very careful when you enable this option to be sure that you are using the correct database. If you expect some strange behavior (like not document found), you may look at this first.

Reference and dereference

You can get the dbref of a document with the *get_dbref()* method. The *dereference()* allow to get a Document from a dbref. You can pass a Document to tell mongokit to what model it should dereferenced:

```
>>> dbref = mydoc.get_dbref()
>>> raw_doc = con.mydb.dereference(dbref) # the result is a regular dict
>>> doc = con.mydb.dereference(dbref, MyDoc) # the result is a MyDoc instance
```

5.2.7 GridFS

MongoKit implements GridFS support and brings some helpers to facilitate the use of relative small files.

Let's create a document `Doc` which have two attachment in GridFS named as *source* and *template*:

```
>>> from mongokit import *
>>> class Doc(Document):
...         structure = {
...             'title':unicode,
...         }
...         gridfs = {'files':['source', 'template']}
```

You might want to be able to add file in gridfs on the fly without knowing their name. The new API allow to add "containers" to gridfs. So, the gridfs declaration look like this:

```
gridfs = {
  'files':['source', 'template'],
  'containers': ['images'],
}
```

As you can see, nothing hard. We just declare our attachment files in the *gridfs* attribute. Filling this attribute will generate an *fs* attribute at runtime. This *fs* attribute is actually an object which deal with GridFS.

```
>>> connection = Connection()
>>> connection.register([Doc])
>>> doc = connection.test.tutorial.Doc()
>>> doc['title'] = u'Hello'
>>> doc.save()
```

Before using gridfs attachment, you have to save the document. This is required as under the hood, mongokit use the document `_id` to link with GridFS files.

The simple way

All gridfs attachments are accessible via the *fs* object. Now, we can fill the `source` and `template`:

```
>>> doc.fs.source = "Hello World !"
>>> doc.fs.template = "My pretty template"
```

And that's it ! By doing this, MongoKit will open a GridFile, fill it with the value, and close it.

Note that you have to be careful to the type : attachments only accept string (Python 2) or bytes (Python 3).

You can read any attachment in a very simple way:

```
>>> doc.fs.source
'Hello World !'
```

You can add any image you want to the container "images":

```
>>> doc.fs.images['image1.png'] = "..."
>>> doc.fs.images['image1.png']
'...'
>>> doc.fs.images['image2.png'] = '...'
```

This is very useful when you want of store a number of file but you don't know their names.

If you have python-magic installed (`sudo easy_install -U python-magic`), the content-type of the file is automatically guessed. To access to it, you have to use the "full way".

new in version 0.5.11

There were many problems with the python-magic support so it has been removed.

If you do not know stored file names, you can list them by iterate:

```
>>> [f.name for f in doc.fs]
['source', 'template']
```

You can list a container as well. The container name is accessible via the *container* attribute:

```
>>> for f in doc.fs.images:
...     print '%s/%s' % (f.container, f.name)
images/image1.png
images/image2.png
```

The full way

While the previous method is very easy, it might not be enougth if you're dealing with very big files or want to use some file related feature (for instance, using seek to not have to load all the file in memory)

You can do that with using the *get_last_version()* method on the `fs` object.

```
>>> f = doc.fs.get_last_version("source")
>>> f.read(10)
```

If you want to create a file and write in it, you can do that with using the *new_file()* method on the `fs` object. The *new_file()* method take the file name and all other properties pymongo accepts:

```
>>> f = doc.fs.new_file('source')
>>> f.write("Hello World again !")
>>> f.close()
```

By supporting PyMongo 1.6 you can use the advanced `with` keyword to handle write operations:

```
>>> with doc.fs.new_file("source") as f:
...     f.write("Hello World again !")
...
```

You can add any image you want to the container "images":

```
>>> f = doc.fs.images.new_file('image1.png')
>>> f.write('...')
>>> f.close()
>>> f = doc.fs.images.get_last_version('image1.png')
>>> f.read(10)
```

All PyMongo API is supported:

```
>>> id = doc.fs.put("Hello World", filename="source")
>>> doc.fs.get(id).read()
'Hello World'
>>> doc.fs.get_last_version("source")
<gridfs.grid_file.GridOut object at 0x1573610>
>>> doc.fs.get_last_version("source").read()
'Hello World'
>>> f = doc.fs.new_file("source")
>>> f.write("New Hello World!")
>>> f.close()
>>> doc.fs.source
'New Hello World!'
>>> new_id = doc.fs.get_last_version("source")._id
>>> doc.fs.delete(new_id)
>>> doc.fs.source
'Hello World'
```

5.3 CRUD Operations

5.3.1 Query

There are two ways to query a collection : raw-data or document-instance .

Getting raw data

Getting raw data is useful when you only want to have one value from your data. This is fast as there's no validation or wrapping. There are four methods to query raw data : *find()* and *find_one()*, *one()* and *find_random()*.

find() and find_one()

find(), and find_one() act like the similar pymongo's methods. Please, see the pymongo documentation.

one()

one() acts like *find()* but will raise a *mongokit.MultipleResultsFound* exception if there is more than one result.

```
>>># Python 3
>>> bp2 = tutorial.BlogPost()
>>> bp2['title'] = 'my second blog post'
>>> bp2['author'] = 'you'
>>> bp2.save()

>>> tutorial.one()
Traceback (most recent call last):
...
MultipleResultsFound: 2 results found

>>> tutorial.one({'title':'my first blog post'})
{'body': None, 'author': myself', 'title': 'my first blog post', 'rank': 0, '_id': ObjectId('4b5ec4b

>>> # Python 2
>>> bp2 = tutorial.BlogPost()
>>> bp2['title'] = u'my second blog post'
>>> bp2['author'] = u'you'
>>> bp2.save()

>>> tutorial.one()
Traceback (most recent call last):
...
MultipleResultsFound: 2 results found
```

```
>>> tutorial.one({'title':'my first blog post'})
{u'body': None, u'author': u'myself', u'title': u'my first blog post', u'rank': 0, u'_id': ObjectId(
```

If no document is found, *one()* returns ^^None^^

find_random()

find_random() will return a random document from the database. This method doesn't take any arguments.

Getting Document instance

There are 5 methods to query your data which return ^^Document^^ instances: *find()*, *find_one()*, *one()*, *fetch()*, *fetch_one()* and *find_random()*. *find()* and *fetch()* return a cursor of collection. A cursor is a container which lazily evaluates the results. A cursor is acting like an iterator. *find_one()*,'one()' and *fetch_one()* return the document itself.

All these methods can take a query as an argument. A query is a simple dict. Check the mongodb and the pymongo documentation for further details.

find()

find() without an argument will return a cursor of all documents from the collection. If a query is passed, it will return a cursor of all documents matching the query.

find() takes the same arguments as the *pymongo.collection.find* method.

```
>>> for post in tutorial.BlogPost.find():
...     print post['title']
my first blog post
my second blog post

>>> for post in tutorial.BlogPost.find({'title':'my first blog post'}):
...     print post['title']
my first blog post
```

find_one()

find_one() acts like *find()* but will return only the first document found. This method takes the same arguments as pymongo's *find_one()* method. Check the pymongo documentation.

one()

one() acts like *find_one()* but will raise a *mongokit.MultipleResultsFound* exception if there is more than one result.

```
>>> # Python 3
>>> tutorial.BlogPost.one()
Traceback (most recent call last):
...
MultipleResultsFound: 2 results found

>>> doc = tutorial.BlogPost.one({'title':'my first blog post'})
>>> doc
{'body': None, 'title': 'my first blog post', 'author': 'myself', 'rank': 0, '_id': ObjectId('4b5ec4
>>> isinstance(doc, BlogPost)
True

>>> # Python 2
>>> tutorial.BlogPost.one()
Traceback (most recent call last):
...
MultipleResultsFound: 2 results found

>>> doc = tutorial.BlogPost.one({'title':'my first blog post'})
>>> doc
{u'body': None, u'title': u'my first blog post', u'author': u'myself', u'rank': 0, u'_id': ObjectId(
>>> isinstance(doc, BlogPost)
True
```

If no document is found, *one()* returns None

fetch()

Unlike *find()*, *fetch()* will return only documents which match the structure of the Document.

```
>>> all_blog_posts = tutorial.BlogPost.fetch()
```

This will return only all blog post (which have 'title', 'body', 'author', 'date_creation', 'rank' as fields). This is an helper for :

```
>>> all_blog_posts = tutorial.BlogPost.find({'body': {'$exists': True}, 'title': {'$exists': True},
```

Note that like with *find()* and *one()*, you can pass advanced queries:

```
>>> my_blog_posts = tutorial.BlogPost.fetch({'author':'myself'})
```

which is equivalent to:

```
>>> all_blog_posts = tutorial.BlogPost.find({'body': {'$exists': True}, 'title': {'$exists': True},
```

fetch_one()

Just like *fetch()* but raise a *mongokit.MultipleResultsFound* exception if there is more than one result.

find_random()

find_random() will return a random document from the database. This method doesn't take other arguments.

5.3.2 Update

Update in Mongokit is as easy as saving an object. Just modify your document and save it:

```
# Python 3
@connection.register
... class MyDoc(Document):
...     structure = {
...         'foo':{
...             'bar':[str],
...             'eggs':{'spam':int},
...         },
...         'bla':str
...     }
```

```
>>> doc = self.col.MyDoc()
>>> doc['_id'] = 3
>>> doc['foo']['bar'] = ['mybar', 'yourbar']
>>> doc['foo']['eggs']['spam'] = 4
>>> doc['bla'] = 'ble'
>>> doc.save()

>>> # Let's modify our doc :
>>> doc['foo']['eggs']['spam'] = 2
>>> doc['bla']= 'bli'
>>> doc.save()
```

```
# Python 2
@connection.register
... class MyDoc(Document):
...    structure = {
...        'foo':{
...            'bar':[unicode],
...            'eggs':{'spam':int},
...        },
...        'bla':unicode
...    }
```

```
>>> doc = self.col.MyDoc()
>>> doc['_id'] = 3
>>> doc['foo']['bar'] = [u'mybar', u'yourbar']
>>> doc['foo']['eggs']['spam'] = 4
>>> doc['bla'] = u'ble'
>>> doc.save()

>>> # Let's modify our doc :
>>> doc['foo']['eggs']['spam'] = 2
>>> doc['bla']= u'bli'
>>> doc.save()
```

Important: You have to be aware that updating a document like that is not atomic. To do so, please read the next section.

Bulk and atomic updates

As Mongokit exposes all the pymongo API, you can use the pymongo's update on collection:

```
>>> con.test.tutorial.update({'title': 'my first blog post'}, {'$set':{'title':u'my very first blog
```

For more information, please look at the pymongo documentation.

reload()

If a document was updated in another thread, it would be necessary to refresh the document to match changes from the database. To do that, use the *reload()* method.

You should know two things before using this method :

- If no _id is set in the document, a KeyError is raised.
- If a document is not saved into the database, the OperationFailure exception is raised.
- using *reload()* will erase all unsaved values !

Example:

```
>>> @connection.register
... class MyDoc(Document):
...     __database__ = 'test'
...     __collection__ = 'tutorial'
...     structure = {
...         'foo':{
...             'eggs':{'spam':int},
...         },
...         'bla':unicode
...     }

>>> doc = connection.MyDoc()
# calling reload() here will raise a KeyError
>>> doc['_id'] = 3
>>> doc['foo']['eggs']['spam'] = 4
>>> doc['bla'] = u'ble'
# calling reload() here will raise an OperationFailure
>>> doc.save()
>>> doc['bla'] = u'bli' # we don't save this change this will be erased
>>> connection.test.tutorial.update({'_id':doc['_id']}, {'$set':{'foo.eggs.spam':2}})
>>> doc.reload()
>>> doc
{'_id': 3, 'foo': {u'eggs': {u'spam': 2}}, 'bla': u'ble'}
```

find_and_modify()

This method allows to return a Document object after or before making an update.

If you call *find_and_modify* on a Collection object, it will return a dict object:

```
>>> d = connection.test.tutorial.find_and_modify({'bla':'ble'}, {'$set':{'foo.eggs.spam':2}})
>>> isinstance(d, MyDoc)
False
>>> isinstance(d, dict)
True
```

If you call *find_and_modify* on a Document object, it will return a Document object:

```
>>> d = connection.MyDoc.find_and_modify({'bla':'ble'}, {'$set':{'foo.eggs.spam':2}})
>>> isinstance(d, MyDoc)
True
```

Please, read the mongodb documentation to learn how to use the *find_and_modify* method.

5.4 Special Features

5.4.1 JSON

While building web application, you might want to create a REST API with JSON support. Then, you may need to convert all your Documents into a JSON format in order to pass it via the REST API. Unfortunately (or fortunately), MongoDB supports field format which is not supported by JSON. This is the case for *datetime* but also for all your *CustomTypes* you may have built and your embedded objects.

`Document` supports the JSON import/export.

to_json()

is a simple method which exports your document into a JSON format:

```
>>> # Python 3
>>> class MyDoc(Document):
...         structure = {
...             "bla":{
...                 "foo":str,
...                 "bar":int,
...             },
...             "spam":[],
...         }
>>> con.register([MyDoc])
>>> mydoc = tutorial.MyDoc()
>>> mydoc['_id'] = 'mydoc'
>>> mydoc["bla"]["foo"] = "bar"
>>> mydoc["bla"]["bar"] = 42
>>> mydoc['spam'] = range(10)
>>> mydoc.save()
>>> json = mydoc.to_json()
>>> json
'{"_id": "mydoc", "bla": {"foo": "bar", "bar": 42}, "spam": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}'

>>> # Python 2
>>> class MyDoc(Document):
...         structure = {
...             "bla":{
...                 "foo":unicode,
...                 "bar":int,
...             },
...             "spam":[],
...         }
>>> con.register([MyDoc])
>>> mydoc = tutorial.MyDoc()
>>> mydoc['_id'] = u'mydoc'
>>> mydoc["bla"]["foo"] = u"bar"
>>> mydoc["bla"]["bar"] = 42
>>> mydoc['spam'] = range(10)
>>> mydoc.save()
>>> json = mydoc.to_json()
>>> json
u'{"_id": "mydoc", "bla": {"foo": "bar", "bar": 42}, "spam": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}'
```

from_json()

To load a JSON string into a `Document`, use the *from_json* class method:

```
>>> # Python 3
>>> class MyDoc(Document):
...     structure = {
...         "bla":{
...             "foo":str,
...             "bar":int,
...         },
...         "spam":[],
...     }
>>> json = '{"_id": "mydoc", "bla": {"foo": "bar", "bar": 42}, "spam": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> mydoc = tutorial.MyDoc.from_json(json)
>>> mydoc
{'_id': 'mydoc', 'bla': {'foo': 'bar', 'bar': 42}, 'spam': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]}

>>> # Python 2
>>> class MyDoc(Document):
...     structure = {
...         "bla":{
...             "foo":unicode,
...             "bar":int,
...         },
...         "spam":[],
...     }
>>> json = '{"_id": "mydoc", "bla": {"foo": "bar", "bar": 42}, "spam": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> mydoc = tutorial.MyDoc.from_json(json)
>>> mydoc
{'_id': 'mydoc', 'bla': {'foo': 'bar', 'bar': 42}, 'spam': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]}
```

Note that *from_json* will take care of all your embedded *Document*s if you used the *to_json()* method to generate the JSON. Indeed, some extra value has to be set : the database and the collection where the embedded document lives. This is added by the *to_json()* method:

```
>>> # Python 3
>>> class EmbedDoc(Document):
...     db_name = "test"
...     collection_name = "mongokit"
...     structure = {
...         "foo":str
...     }
>>> class MyDoc(Document):
...    db_name = "test"
...    collection_name = "mongokit"
...    structure = {
...        "doc":{
...            "embed":EmbedDoc,
...        },
...    }
...    use_autorefs = True
>>> con.register([EmbedDoc, MyDoc])

>>> # Python 2
>>> class EmbedDoc(Document):
...     db_name = "test"
...     collection_name = "mongokit"
...     structure = {
```

```
...         "foo":unicode
...     }
>>> class MyDoc(Document):
...    db_name = "test"
...    collection_name = "mongokit"
...    structure = {
...        "doc":{
...            "embed":EmbedDoc,
...        },
...    }
...    use_autorefs = True
>>> con.register([EmbedDoc, MyDoc])
```

Let's create an embedded doc:

```
>>> # Python 3
>>> embed = tutorial.EmbedDoc()
>>> embed['_id'] = "embed"
>>> embed['foo'] = "bar"
>>> embed.save()

>>> # Python 2
>>> embed = tutorial.EmbedDoc()
>>> embed['_id'] = u"embed"
>>> embed['foo'] = u"bar"
>>> embed.save()
```

and embed this doc to another doc:

```
>>> # Python 3
>>> mydoc = tutorial.MyDoc()
>>> mydoc['_id'] = u'mydoc'
>>> mydoc['doc']['embed'] = embed
>>> mydoc.save()

>>> # Python 2
>>> mydoc = tutorial.MyDoc()
>>> mydoc['_id'] = 'mydoc'
>>> mydoc['doc']['embed'] = embed
>>> mydoc.save()
```

Now let's see how the generated json looks like:

```
>>> json = mydoc.to_json()
>>> json
u'{"doc": {"embed": {"_collection": "tutorial", "_database": "test", "_id": "embed", "foo": "bar"}},
```

As you can see, two new fields have been added : *_collection* and *_database* which represent respectively the collection and the database where the embedded doc has been saved. That information is needed to generate the embedded document. These are removed when calling the *from_json()* method:

```
>>> # Python 3
>>> mydoc = tutorial.MyDoc.from_json(json)
>>> mydoc
{u'doc': {u'embed': {u'_id': u'embed', u'foo': u'bar'}}, u'_id': u'mydoc'}

>>> # Python 2
>>> mydoc = tutorial.MyDoc.from_json(json)
>>> mydoc
```

```
{'doc': {'embed': {'_id': 'embed', 'foo': 'bar'}}, '_id': 'mydoc'}
```

An the embedded document is an instance of EmbedDoc:

```
>>> isinstance(mydoc['doc']['embed'], EmbedDoc)
True
```

ObjectId support

from_json() can detect if the *_id* is an `ObjectId` instance or a simple string. When you serialize an object with `ObjectId` instance to json, the generated json object looks like this:

```
'{"_id": {"$oid": "..."}, ...}'
```

The "$oid" field is added to tell *from_json()* that '_id' is an `ObjectId` instance. The same happens with embedded docs:

```
>>> mydoc = tutorial.MyDoc()
>>> mydoc['doc']['embed'] = embed
>>> mydoc.save()
>>> mydoc.to_json()
{'doc': {'embed': {u'_id': ObjectId('4b5ec45090bce737cb000002'), u'foo': u'bar'}}, '_id': ObjectId('4
```

5.4.2 Migration

Let's say we have created a blog post which look like this:

```
# Python 3
>>> from mongokit import *
>>> con = Connection()
... class BlogPost(Document):
...     structure = {
...         "blog_post":{
...             "title": str,
...             "created_at": datetime,
...             "body": str,
...         }
...     }
...     default_values = {'blog_post.created_at':datetime.utcnow()}

>>> # Python 2
>>> from mongokit import *
>>> con = Connection()
... class BlogPost(Document):
...     structure = {
...         "blog_post":{
...             "title": unicode,
...             "created_at": datetime,
...             "body": unicode,
...         }
...     }
...     default_values = {'blog_post.created_at':datetime.utcnow()}
```

Let's create some blog posts:

```
>>> for i in range(10):
...     con.test.tutorial.BlogPost({'title':u'hello %s' % i, 'body': u'I the post number %s' % i}).sa
```

Now, development goes on and we add a 'tags' field to our *BlogPost*:

```
# Python 3
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": str,
            "created_at": datetime,
            "body": str,
            "tags":  [str],
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow()}

# Python 2
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": unicode,
            "created_at": datetime,
            "body": unicode,
            "tags":  [unicode],
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow()}
```

We're gonna be in trouble when we'll try to save the fetched document because the structures don't match:

```
>>> blog_post = con.test.tutorial.BlogPost.find_one()
>>> blog_post['blog_post']['title'] = u'Hello World'
>>> blog_post.save()
Traceback (most recent call last):
    ...
StructureError: missed fields : ['tags']
```

If we want to fix this issue, we have to add the 'tags' field manually to all *BlogPost* in the database:

```
>>> con.test.tutorial.update({'blog_post':{'$exists':True}, 'blog_post.tags':{'$exists':False}},
...    {'$set':{'blog_post.tags':[]}}, multi=True)
```

and now we can save our blog_post:

```
>>> blog_post.reload()
>>> blog_post['blog_post']['title'] = u'Hello World'
>>> blog_post.save()
```

Lazy migration

Important: You cannot use this feature if *use_schemaless* is set to True

Mongokit provides a convenient way to set migration rules and apply them lazily. We will explain how to do that using the previous example.

Let's create a *BlogPostMigration* which inherits from *DocumentMigration*:

```
class BlogPostMigration(DocumentMigration):
    def migration01__add_tags_field(self):
        self.target = {'blog_post':{'$exists':True}, 'blog_post.tags':{'$exists':False}}
        self.update = {'$set':{'blog_post.tags':[]}}
```

How does it work? All migration rules are simple methods on the *BlogPostMigration*. They must begin with *migration* and be numbered (so they can be applied in certain order). The rest of the name should describe the rule. Here, we create our first rule (*migration01*) which adds the 'tags' field to our *BlogPost*.

Then you must set two attributes : *self.target* and *self.update*. There's both mongodb regular query.

self.target will tell mongokit which document will match this rule. Migration will be applied to every document matching this query.

self.update is a mongodb update query with modifiers. This will describe what updates should be applied to the matching document.

Now that our *BlogPostMigration* is created, we have to tell Mongokit to what document these migration rules should be applied. To do that, we have to set the *migration_handler* in *BlogPost*:

```
# Python 3
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": unicode,
            "created_at": datetime,
            "body": unicode,
            "tags": [unicode],
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow}
    migration_handler = BlogPostMigration

# Python 2
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": unicode,
            "created_at": datetime,
            "body": unicode,
            "tags": [unicode],
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow}
    migration_handler = BlogPostMigration
```

Each time an error is raised while validating a document, migration rules are applied to the object and the document is reloaded.

Caution: If *migration_handler* is set then *skip_validation* is deactivated. Validation must be on to allow lazy migration.

Bulk migration

Lazy migration is useful if you have many documents to migrate, because update will lock the database. But sometimes you might want to make a migration on few documents and you don't want slow down your application with validation. You should then use bulk migration.

Bulk migration works like lazy migration but *DocumentMigration* method must start with *allmigration*. Because lazy migration adds document *_id* to *self.target*, with bulk migration you should provide more information on *self.target*. Here's an example of bulk migration, where we finally wan't to remove the *tags* field from *BlogPost*:

```
# Python 3
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": unicode,
            "creation_date": datetime,
            "body": unicode,
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow}

# Python 2
class BlogPost(Document):
    structure = {
        "blog_post":{
            "title": unicode,
            "creation_date": datetime,
            "body": unicode,
        }
    }
    default_values = {'blog_post.created_at':datetime.utcnow}
```

Note that we don't need to add the *migration_handler*, it is required only for lazy migration.

Let's edit the *BlogPostMigration*:

```
class BlogPostMigration(DocumentMigration):
    def allmigration01_remove_tags(self):
        self.target = {'blog_post.tags':{'$exists':True}}
        self.update = {'$unset':{'blog_post.tags':[]}}
```

To apply the migration, instantiate the *BlogPostMigration* and call the *migrate_all* method:

```
>>> migration = BlogPostMigration(BlogPost)
>>> migration.migrate_all(collection=con.test.tutorial)
```

Note: Because *migration_** methods are not called with *migrate_all()*, you can mix *migration_** and *allmigration_** methods.

Migration status

Once all your documents have been migrated, some migration rules could become deprecated. To know which rules are deprecated, use the *get_deprecated* method:

```
>>>> migration = BlogPostMigration(BlogPost)
>>> migration.get_deprecated(collection=con.test.tutorial)
{'deprecated':['allmigration01__remove_tags'], 'active':['migration02__rename_created_at']}
```

Here we can remove the rule *allmigration01__remove_tags*.

Advanced migration

Lazy migration

Sometimes we might want to build more advanced migration. For instance, say you want to copy a field value into another field, you can have access to the current doc value via *self.doc*. In the following example, we want to add an *update_date* field and copy the *creation_date* value into it:

```
class BlogPostMigration(DocumentMigration):
    def migration01__add_update_field_and_fill_it(self):
        self.target = {'blog_post.update_date':{'$exists':False}, 'blog_post':{'$exists':True}}
        self.update = {'$set':{'blog_post.update_date': self.doc['blog_post']['creation_date']}}
```

Advanced and bulk migration

If you want to do the same thing with bulk migration, things are a little different:

```
class BlogPostMigration(DocumentMigration):
    def allmigration01__add_update_field_and_fill_it(self):
        self.target = {'blog_post.update_date':{'$exists':False}, 'blog_post':{'$exists':True}}
        if not self.status:
            for doc in self.collection.find(self.target):
                self.update = {'$set':{'blog_post.update_date': doc['blog_post']['creation_date']}}
                self.collection.update(self.target, self.update, multi=True, safe=True)
```

In this example, the method *allmigration01__add_update_field_and_fill_it* will directly modify the database and will be called by *get_deprecated()*. But calling *get_deprecated()* should not arm the database so, we need to specify what portion of the code must be ignored when calling *get_deprecated()*. This explains the second line.

5.4.3 Paginator

Implementing pagination in the project is made easy with `mongokit.paginator.Paginator`. `Paginator` actually converts query-result-cursor into Paginator object and provides useful properties on it.

Using *Paginator* is consists of following two logical steps:

1. Importing *paginator.Paginator* module.
2. Applying it on your query-result-cursor.

Lets apply this steps with following detailed example.

A detailed Example:

Consider following as a sample model class:

```
>>> # Python 3
>>> from mongokit import Document, Connection
>>> connection = Connection()
...
... @connection.register
... class Wiki(Document):
...
...     __collection__ = 'wiki'
...     __database__ = 'db_test_pagination'
```

```
...
...     structure = {
...         "name": str,  # name of wiki
...         "description": str,  # content of wiki
...         "created_by": str,  # username of user
...     }
...
...     required_fields = ['name', 'created_by']

>>> # Python 2
>>> from mongokit import Document, Connection
>>> connection = Connection()
...
... @connection.register
... class Wiki(Document):
...
...     __collection__ = 'wiki'
...     __database__ = 'db_test_pagination'
...
...     structure = {
...         "name": unicode,  # name of wiki
...         "description": unicode,  # content of wiki
...         "created_by": basestring,  # username of user
...     }
...
...     required_fields = ['name', 'created_by']
```

Now let's consider that you have created 55 instances of class Wiki. And while querying you are getting all the instances in a query-result-cursor or resultant cursor.

```
>>> wiki_collection = connection['db_test_pagination']
>>> total_wikis = wiki_collection.Wiki.find()
>>> total_wikis.count()
... 55
```

Now let's paginate the resultant cursor: `total_wikis`

As stated previously, we will first import the `Paginator` and then apply pagination on the resultant cursor.

```
>>> from mongokit.paginator import Paginator
>>> page_no = 2  # page number
>>> no_of_objects_pp = 10  # total no of objects or items per page
```

Keyword arguments required for `Paginator` class are as follows:

1. `cursor` – Cursor of a returned query (`total_wikis` in our example)
2. `page` – The page number requested (`page_no` in our example)
3. `limit` – The number of items per page (`no_of_objects_pp` in our example)

```
    >>> paged_wiki = Paginator(total_wikis, page_no, no_of_objects_pp)
```

We had applied the pagination on `total_wikis` cursor and stored it's result in `paged_wiki`, which is a Paginated object.

Note The cursor (`total_wikis`) which we passed as an argument for `Paginator`, also gets limit to `no_of_objects_pp` (`10` in our case). And looping it would loop for `no_of_objects_pp` (`10`) times.

Paginated object properties:

Let's move ahead and try properties on `paged_wiki`. There are total of 11 properties provided by mongokit, for the Paginated object. The properties that we can apply on `paged_wiki` are as follows:

property-1: **items** – *Returns the paginated Cursor object.*

Above code will loop for 10 times to print the name of objects.

Property-2: **is_paginated** – *Boolean value determining if the cursor has multiple pages*

```
>>> paged_wiki.is_paginated
... True
```

Property-3: **start_index** – *int index of the first item on the requested page*

```
>>> paged_wiki.start_index
... 11
```

As the no. of items per page is `10`, we got the result of second page's, starting item's index as `11`.

Property-4: **end_index** – *int index of the last item on the requested page*

```
>>> paged_wiki.end_index
... 20
```

As the no. of items per page is `10`, we got the result of second page's, ending item's index as `20`.

Property-5: **current_page** – *int page number of the requested page*

```
>>> paged_wiki.current_page
... 2
```

Property-6: **previous_page** – *int page number of previous page with respect to current requested page*

```
>>> paged_wiki.previous_page
... 1
```

Property-7: **next_page** – *int page number of next page with respect to current requested page*

```
>>> paged_wiki.next_page
... 3
```

Property-8: **has_next** – *True or False if the Cursor has a next page*

```
>>> paged_wiki.has_next
... True
```

Property-9: **has_previous** – *True or False if the Cursor has a previous page*

```
>>> paged_wiki.has_previous
... True
```

Property-10: **page_range** – *list of the all page numbers (ascending order) in a list format*

```
>>> paged_wiki.page_range
... [1, 2, 3, 4, 5, 6]
```

Property-11: **num_pages** – *int of the total number of pages*

```
>>> paged_wiki.num_pages
... 6
```

Property-12: **count** – *int total number of items on the cursor*

```
>>> paged_wiki.count
... 55
```

It's same as that of `total_wikis.count()`

5.5 Frameworks Support

5.5.1 Django

5.5.2 Flask

5.5.3 Pylons

You can very easily interface Mongokit with Pylons. This tutorial explains how to do this. Note that there's a lot of ways to do the same thing. If you found another better solution, please contact me, I'll update this tutorial.

Write all your models in `model/`. In the `model/__init__.py`, import all the module you want to register and then add them to a list called *register_models*.

Example of `model/__init__.py`:

```
from blog import Blog
from blogpost import BlogPost

register_models = [Blog, BlogPost]
```

Then go to the `lib/app_globals.py` and edit this file so it look like this:

```
from pylons import config
from <appname>.models import register_models
from mongokit import *

class Globals(object):

    """Globals acts as a container for objects available throughout the
    life of the application

    """

    def __init__(self):
        """One instance of Globals is created during application
        initialization and is available during requests via the
        'app_globals' variable

        """
        self.connection = Connection(
          host = config['db_host'],
          port = int(config['db_port']),
        )
        self.db = self.connection[config['db_name']]
        self.connection.register(register_models)
```

In this file, we create the connection (and optionally the db if we use one) and then we register all our models.

Now, you can access the connection via the pylons.config :

config['pylons.app_globals'].connection

A more convenient way is to add the connection to the BaseController so you can access it just with `self.connection`. The file `lib/base.py` has to look like this:

```
from pylons.controllers import WSGIController
from pylons import config
import pylons

class BaseController(WSGIController):

    connection = config['pylons.app_globals'].connection

    def __call__(self, environ, start_response):
        """Invoke the Controller"""
        # WSGIController.__call__ dispatches to the Controller method
        # the request is routed to. This routing information is
        # available in environ['pylons.routes_dict']
        return WSGIController.__call__(self, environ, start_response)
```

5.6 Development

5.6.1 Tests

Software Testing is necessary because we all make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. To avoid that we write tests.

Package Building/Testing

Package building/testing is what Travis CI does. It would be better and easier to have your own travis for your fork. But if you want to almost simulate what it does locally here is steps:

```
$ # change master to the branch you want to test, also don't forget to change the username
$ git clone --depth=50 --branch=master git://github.com/username/mongokit.git
$ make setup
$ make test
```

Note: It's quite important to have appropriate Python environment. If your python version on your system or virtual environment is 2.7 it doesn't tests for other versions of python and you should create different enthronement your self to run the tests.

Tox Automated Test

Another way to run tests is to use Tox. The advantage of using tox is its automation of creating different virtual environment for python versions that are described in `tox.ini`. For MongoKit we have Python 2.7, Python 3.3, and Python 3.4.

Make sure you have installed Tox. And if you haven't:

```
pip install tox
```

To run all the tests in all defined environments simply run:

```
$ tox
```

and to run test in a specified environment use:

```
$ tox -e py34
```

Note: py34 is defined in `tox.ini`. Other options are py33 and py27.

Nose Test

But if you ant run tests partially or run a single test Nose is the way to go.'Nose'_ extends unittest to make testing easier. To run all the tests use:

```
$ nosetests
```

and if you wanted to run test for specific feature/file use:

```
$ nosetests test/test_api.py
```

Note: For further instructions please view selecting test.

5.6.2 Documentation

As a developer, it's always important to have reliable documentation to guide your work. Here is brief overview of how to create local documentation server:

Local Documentation Server

While you are writing documentation or you have slow internet connection it would be nice to have local documentation server.

To create local documentation server clone the repo:

```
$ git clone https://github.com/namlook/mongokit.git
$ cd mongokit
```

or you can download the zip file:

```
$ wget https://github.com/namlook/mongokit/archive/master.zip
$ unzip master.zip
```

and to compile:

```
$ cd mongokit/docs
$ make html
sphinx-build -b html -d build/doctrees   source build/html
Making output directory...
...
build succeeded, 51 warnings.
```

Sphinx would create a build/html directory, go into it:

```
$ cd build/html
```

Run python http server

```
$ # Python 3
$ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...

$ # Python 2
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Open your browser and go to http://localhost:8000 or http://127.0.0.1:8000 .

Note: It would be better to open a new terminal as documentation http server, that way you can see changes simultaneously each time html directory was updated.

If some unwanted results was produced, the quick fix would be deleting the cached build and remaking it:

```
$ cd ../.. && pwd
/home/user/mongokit/docs
$ make clean
rm -rf build/*
$ make html
sphinx-build -b html -d build/doctrees   source build/html
Making output directory...
...
build succeeded, 51 warnings.
```

See also:

It worth mentioning a very nice package called sphinx-autobuild which automates all the steps described. What it does is that it watches a Sphinx directory and rebuild the documentation when a change is detected. Also includes a livereload enabled web server.

5.7 API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

5.7.1 Auth

class `mongokit.auth.`**`User`**`(`**args*, ***kwargs*`)`

`authorized_types` = [<class 'dict'>, <class 'bson.binary.Binary'>, <class 'bytes'>, <class 'datetime.datetime'>, <class

`del_email``()`

`del_login``()`

`del_password``()`

`email`

`get_email``()`

`get_login``()`

get_password()
Return the password hashed

login

password
Return the password hashed

required_fields = ['user.password', 'user.email']

save(**args*, ***kwargs*)

set_email(*email*)

set_login(*login*)

set_password(*password*)
Hash password on the fly

structure = {'_id': <class 'str'>, 'user': {'email': <class 'str'>, 'login': <class 'str'>, 'password': <class 'str'>}}

use_dot_notation = True

verify_password(*password*)
Check the password against existing credentials

5.7.2 Collection

class mongokit.collection.**Collection**(**args*, ***kwargs*)

find(**args*, ***kwargs*)
Query the database.

The *spec* argument is a prototype document that all results must match. For example:

```
>>> db.test.find({"hello": "world"})
```

only matches documents that have a key "hello" with value "world". Matches can have other keys *in addition* to "hello". The *fields* argument is used to specify a subset of fields that should be included in the result documents. By limiting results to a certain subset of fields you can cut down on network traffic and decoding time.

Raises `TypeError` if any of the arguments are of improper type. Returns an instance of `Cursor` corresponding to this query.

Parameters

- *spec* (optional): a SON object specifying elements which must be present for a document to be included in the result set
- *fields* (optional): a list of field names that should be returned in the result set or a dict specifying the fields to include or exclude. If *fields* is a list "_id" will always be returned. Use a dict to exclude fields from the result (e.g. fields={'_id': False}).
- *skip* (optional): the number of documents to omit (from the start of the result set) when returning the results
- *limit* (optional): the maximum number of results to return
- *timeout* (optional): if True (the default), any returned cursor is closed by the server after 10 minutes of inactivity. If set to False, the returned cursor will never time out on the server. Care should be taken to ensure that cursors with timeout turned off are properly closed.

- *snapshot* (optional): if True, snapshot mode will be used for this query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution. For details, see the snapshot documentation.
- *tailable* (optional): the result of this find call will be a tailable cursor - tailable cursors aren't closed when the last data is retrieved but are kept open and the cursors location marks the final document's position. if more data is received iteration of the cursor will continue from the last document received. For details, see the tailable cursor documentation.
- *sort* (optional): a list of (key, direction) pairs specifying the sort order for this query. See `sort()` for details.
- *max_scan* (optional): limit the number of documents examined when performing the query
- *as_class* (optional): class to use for documents in the query result (default is `document_class`)
- *slave_okay* (optional): if True, allows this query to be run against a replica secondary.
- *await_data* (optional): if True, the server will block for some extra time before returning, waiting for more data to return. Ignored if *tailable* is False.
- *partial* (optional): if True, mongos will return partial results if some shards are down instead of returning an error.
- *manipulate*: (optional): If True (the default), apply any outgoing SON manipulators before returning.
- *network_timeout* (optional): specify a timeout to use for this query, which will override the `MongoClient`-level default
- *read_preference* (optional): The read preference for this query.
- *tag_sets* (optional): The tag sets for this query.
- *secondary_acceptable_latency_ms* (optional): Any replica-set member whose ping time is within secondary_acceptable_latency_ms of the nearest member may accept reads. Default 15 milliseconds. **Ignored by mongos** and must be configured on the command line. See the localThreshold option for more information.

Note: The *manipulate* parameter may default to False in a future release.

Note: The *max_scan* parameter requires server version **>= 1.5.1**

New in version 2.3: The *tag_sets* and *secondary_acceptable_latency_ms* parameters.

New in version 1.11+: The *await_data*, *partial*, and *manipulate* parameters.

New in version 1.8: The *network_timeout* parameter.

New in version 1.7: The *sort*, *max_scan* and *as_class* parameters.

Changed in version 1.7: The *fields* parameter can now be a dict or any iterable in addition to a list.

New in version 1.1: The *tailable* parameter.

added by mongokit::

- *wrap* (optional): a class object used to wrap documents in the query result

find_and_modify(**args*, ***kwargs*)

Update and return an object.

This is a thin wrapper around the findAndModify command. The positional arguments are designed to match the first three arguments to `update()` however most options should be passed as named parameters. Either *update* or *remove* arguments are required, all others are optional.

Returns either the object before or after modification based on *new* parameter. If no objects match the *query* and *upsert* is false, returns `None`. If upserting and *new* is false, returns `{}`.

If the full_response parameter is `True`, the return value will be the entire response object from the server, including the 'ok' and 'lastErrorObject' fields, rather than just the modified object. This is useful mainly because the 'lastErrorObject' document holds information about the command's execution.

Parameters

- *query*: filter for the update (default `{}`)
- *update*: see second argument to `update()` (no default)
- *upsert*: insert if object doesn't exist (default `False`)
- *sort*: a list of (key, direction) pairs specifying the sort order for this query. See `sort()` for details.
- *full_response*: return the entire response object from the server (default `False`)
- *remove*: remove rather than updating (default `False`)
- *new*: return updated rather than original object (default `False`)
- *fields*: see second argument to `find()` (default all)
- ***kwargs*: any other options the findAndModify command supports can be passed here.

Note: Requires server version **>= 1.3.0**

Changed in version 2.5: Added the optional full_response parameter

Changed in version 2.4: Deprecated the use of mapping types for the sort parameter

New in version 1.10.

added by mongokit::

- *wrap* (optional): a class object used to wrap

documents in the query result

find_fulltext(*search*, ***kwargs*)

Executes a full-text search. Additional parameters may be passed as keyword arguments.

find_random()

return one random document from the collection

get_from_id(*id*)

return the document which has the id

one(**args*, ***kwargs*)

5.7.3 Connection

class `mongokit.connection.`**`CallableMixin`**
brings the callable method to a Document. usefull for the connection's register method

class `mongokit.connection.`**`Connection`**(**args*, ***kwargs*)

`mongokit.connection.`**`MongoClient`**
alias of `Connection`

class `mongokit.connection.`**`MongoKitConnection`**(**args*, ***kwargs*)

`register`(*obj_list*)

`mongokit.connection.`**`MongoReplicaSetClient`**
alias of `ReplicaSetConnection`

class `mongokit.connection.`**`ReplicaSetConnection`**(**args*, ***kwargs*)

5.7.4 Cursor

class `mongokit.connection.`**`CallableMixin`**
brings the callable method to a Document. usefull for the connection's register method

class `mongokit.connection.`**`Connection`**(**args*, ***kwargs*)

`mongokit.connection.`**`MongoClient`**
alias of `Connection`

class `mongokit.connection.`**`MongoKitConnection`**(**args*, ***kwargs*)

`register`(*obj_list*)

`mongokit.connection.`**`MongoReplicaSetClient`**
alias of `ReplicaSetConnection`

class `mongokit.connection.`**`ReplicaSetConnection`**(**args*, ***kwargs*)

5.7.5 Database

class `mongokit.database.`**`Database`**(**args*, ***kwargs*)

`dereference`(*dbref*, *model=None*)

5.7.6 Document

class `mongokit.document.`**`Document`**(*doc=None*, *gen_skel=True*, *collection=None*, *lang='en'*, *fallback_lang='en'*)

`atomic_save` **= False**

`authorized_types` **= [<class 'dict'>, <class 'bson.binary.Binary'>, <class 'bytes'>, <class 'datetime.datetime'>, <class**

`delete`()
delete the document from the collection from his _id.

fetch (*spec=None*, **args*, ***kwargs*)
return all document which match the structure of the object *fetch()* takes the same arguments than the the pymongo.collection.find method.

The query is launch against the db and collection of the object.

fetch_one (**args*, ***kwargs*)
return one document which match the structure of the object *fetch_one()* takes the same arguments than the the pymongo.collection.find method.

If multiple documents are found, raise a MultipleResultsFound exception. If no document is found, return None

The query is launch against the db and collection of the object.

find (**args*, ***kwargs*)
Query the database.

The *spec* argument is a prototype document that all results must match. For example if self si called MyDoc:

```
>>> mydocs = db.test.MyDoc.find({"hello": "world"})
```

only matches documents that have a key "hello" with value "world". Matches can have other keys *in addition* to "hello". The *fields* argument is used to specify a subset of fields that should be included in the result documents. By limiting results to a certain subset of fields you can cut down on network traffic and decoding time.

mydocs is a cursor which yield MyDoc object instances.

See pymongo's documentation for more details on arguments.

find_and_modify (**args*, ***kwargs*)
Update and return an object.

find_fulltext (*search*, ***kwargs*)
Executes a full-text search. Additional parameters may be passed as keyword arguments.

find_one (**args*, ***kwargs*)
Get the first object found from the database.

See pymongo's documentation for more details on arguments.

find_random ()
return one random document from the collection

force_autorefs_current_db = False

from_json (*json*)
convert a json string and return a SchemaDocument

classmethod generate_index (*collection*)
generate indexes from `indexes` class-attribute

supports additional index-creation-keywords supported by pymongos `ensure_index`.

get_dbref ()
return a pymongo DBRef instance related to the document

get_from_id (*id*)
return the document which has the id

get_size ()
return the size of the underlying bson object

gridfs = []

indexes = []

migrate(*safe=True*, *_process_to_bson=True*)
migrate the document following the migration_handler rules

safe : if True perform a safe update (see pymongo documentation for more details

migration_handler = None

one(**args*, ***kwargs*)
one() act like *find()* but will raise a *mongokit.MultipleResultsFound* exception if there is more than one result.

If no document is found, *one()* returns *None*

reload()
allow to refresh the document, so after using update(), it could reload its value from the database.

Be careful : reload() will erase all unsaved values.

If no _id is set in the document, a KeyError is raised.

save(*uuid=False*, *validate=None*, *safe=True*, **args*, ***kwargs*)
save the document into the db.

if uuid is True, a uuid4 will be automatically generated else, the bson.ObjectId will be used.

If validate is True, the *validate* method will be called before saving. Not that the *validate* method will be called *before* the uuid is generated.

save() follow the pymongo.collection.save arguments

skip_validation = False

to_json()
convert the document into a json string and return it

to_json_type()
convert all document field into json type and return the new converted object

type_field = '_type'

use_autorefs = False

validate(*auto_migrate=False*)

class mongokit.document.**DocumentProperties**

class mongokit.document.**R**(*doc*, *connection*, *fallback_database=None*)
CustomType to deal with autorefs documents

mongo_type
alias of `DBRef`

python_type
alias of `Document`

to_bson(*value*)

to_python(*value*)

5.7.7 Grid

class `mongokit.grid.`**`FS`**(*obj*)

`get_version`(*filename*, *version=-1*, ***kwargs*)
Get a file from GridFS by `"filename"` or metadata fields.

Returns a version of the file in GridFS whose filename matches *filename* and whose metadata fields match the supplied keyword arguments, as an instance of `GridOut`.

Version numbering is a convenience atop the GridFS API provided by MongoDB. If more than one file matches the query (either by *filename* alone, by metadata fields, or by a combination of both), then version `-1` will be the most recently uploaded matching file, `-2` the second most recently uploaded, etc. Version `0` will be the first version uploaded, `1` the second version, etc. So if three versions have been uploaded, then version `0` is the same as version `-3`, version `1` is the same as version `-2`, and version `2` is the same as version `-1`.

Raises `NoFile` if no such version of that file exists.

An index on `{filename: 1, uploadDate: -1}` will automatically be created when this method is called the first time.

Parameters

- *filename*: `"filename"` of the file to get, or *None*
- *version* (optional): version of the file to get (defaults to -1, the most recent version uploaded)
- ***kwargs* (optional): find files by custom metadata.

Changed in version 1.11: *filename* defaults to None;

New in version 1.11: Accept keyword arguments to find files by custom metadata.

New in version 1.9.

`new_file`(*filename*)

`put`(*data*, ***kwargs*)

class `mongokit.grid.`**`FSContainer`**(*container_name*, *obj*)

5.7.8 Helpers

class `mongokit.helpers.`**`DotCollapsedDict`**(*passed_dict*, *remove_under_type=False*, *reference=None*)
A special dictionary constructor that take a dict and provides a dot collapsed dict:

```
>>> DotCollapsedDict({'a':{'b':{'c':{'d':3}, 'e':5}, "g":2}, 'f':6})
{'a.b.c.d': 3, 'a.b.e': 5, 'a.g': 2, 'f': 6}
```

```
>>> DotCollapsedDict({'bla':{'foo':{unicode:{"bla":3}}, 'bar':'egg'}})
{'bla.foo.$unicode.bla': 3, 'bla.bar': "egg"}
```

```
>>> DotCollapsedDict({'bla':{'foo':{unicode:{"bla":3}}, 'bar':'egg'}}, remove_under_type=True)
{'bla.foo':{}, 'bla.bar':unicode}
```

```
>>> dic = {'bar':{'foo':3}, 'bla':{'g':2, 'h':3}}
>>> DotCollapsedDict(dic, reference={'bar.foo':None, 'bla':{'g':None, 'h':None}})
{'bar.foo':3, 'bla':{'g':2, 'h':3}}
```

class mongokit.helpers.**DotExpandedDict**(*key_to_list_mapping*)
A special dictionary constructor that takes a dictionary in which the keys may contain dots to specify inner dictionaries. It's confusing, but this example should make sense.

```
>>> d = DotExpandedDict({'person.1.firstname': ['Simon'],          'person.1.lastname': ['Willi
>>> d
{'person': {'1': {'lastname': ['Willison'], 'firstname': ['Simon']},
'2': {'lastname': ['Holovaty'], 'firstname': ['Adrian']}}}
>>> d['person']
{'1': {'lastname': ['Willison'], 'firstname': ['Simon']}, '2': {'lastname': ['Holovaty'], 'first
>>> d['person']['1']
{'lastname': ['Willison'], 'firstname': ['Simon']}
```

# Gotcha: Results are unpredictable if the dots are "uneven": >>> DotExpandedDict({'c.1': 2, 'c.2': 3, 'c': 1}) {'c': 1}

class mongokit.helpers.**DotedDict**(*doc=None*, *warning=False*)
Dot notation dictionary access

mongokit.helpers.**fromtimestamp**(*epoch_date*)
convert a float since epoch to a datetime object

class mongokit.helpers.**i18nDotedDict**(*dic*, *doc*)
Dot notation dictionary access with i18n support

mongokit.helpers.**totimestamp**(*value*)
convert a datetime into a float since epoch

5.7.9 Master Slave Connection

Master-Slave integration with for MongoKit Andreas Jung, info@zopyx.com (same license as Mongokit)

class mongokit.master_slave_connection.**MasterSlaveConnection**(*master*, *slaves=[]*)
Master-Slave support for MongoKit

5.7.10 Migration

class mongokit.migration.**DocumentMigration**(*doc_class*)

clean()

get_deprecated(*collection*)

migrate(*doc*, *safe=True*)
migrate the doc through all migration process

migrate_all(*collection*, *safe=True*)

validate_update(*update_query*)

5.7.11 Migration

exception mongokit.mongo_exceptions.**AutoReferenceError**

exception mongokit.mongo_exceptions.**BadIndexError**

exception mongokit.mongo_exceptions.**ConnectionError**

exception `mongokit.mongo_exceptions.`**`EvalException`**

exception `mongokit.mongo_exceptions.`**`MaxDocumentSizeError`**

exception `mongokit.mongo_exceptions.`**`MongoAuthException`**

exception `mongokit.mongo_exceptions.`**`MultipleResultsFound`**

exception `mongokit.mongo_exceptions.`**`OptionConflictError`**

exception `mongokit.mongo_exceptions.`**`UpdateQueryError`**

5.7.12 Operators

class `mongokit.operators.`**`IS`**(**args*)

> **`repr`** = 'is'
>
> **`validate`**(*value*)

class `mongokit.operators.`**`NOT`**(**args*)

> **`repr`** = 'not'
>
> **`validate`**(*value*)

class `mongokit.operators.`**`OR`**(**args*)

> **`repr`** = 'or'
>
> **`validate`**(*value*)

class `mongokit.operators.`**`SchemaOperator`**(**args*)

> **`repr`** = None
>
> **`validate`**(*value*)

5.7.13 Paginator

class `mongokit.paginator.`**`Paginator`**(*cursor*, *page=1*, *limit=10*)

> Provides pagination on a Cursor object
>
> Keyword arguments: cursor – Cursor of a returned query page – The page number requested limit – The number of items per page
>
> Properties: items – Returns the paginated Cursor object is_paginated – Boolean value determining if the cursor has multiple pages start_index – int index of the first item on the requested page end_index – int index of the last item on the requested page current_page – int page number of the requested page previous_page– int page number of the previous page w.r.t. current requested page next_page – int page number of the next page w.r.t. current requested page has_next – True or False if the Cursor has a next page has_previous – True or False if the Cursor has a previous page page_range – list of page numbers num_pages – int of the number of pages count – int total number of items on the cursor
>
> **`count`**
>
> **`current_page`**
>
> **`end_index`**

has_next

has_previous

is_paginated

items

next_page

num_pages

page_range

previous_page

start_index

5.7.14 Schema Document

exception `mongokit.schema_document.`**`AuthorizedTypeError`**

exception `mongokit.schema_document.`**`BadKeyError`**

class `mongokit.schema_document.`**`CustomType`**

init_type = None

mongo_type = None

python_type = None

to_bson (*value*)
convert type to a mongodb type

to_python (*value*)
convert type to a mongodb type

validate (*value*, *path*)
This method is optional. It add a validation layer. This method is been called in Document.validate()

value: the value of the field path: the field name (ie, 'foo' or 'foo.bar' if nested)

exception `mongokit.schema_document.`**`DefaultFieldTypeError`**

class `mongokit.schema_document.`**`DotCollapsedDict`** (*passed_dict*, *remove_under_type=False*, *reference=None*)
A special dictionary constructor that take a dict and provides a dot collapsed dict:

```
>>> DotCollapsedDict({'a':{'b':{'c':{'d':3}, 'e':5}, "g":2}, 'f':6})
{'a.b.c.d': 3, 'a.b.e': 5, 'a.g': 2, 'f': 6}
```

```
>>> DotCollapsedDict({'bla':{'foo':{unicode:{"bla":3}}, 'bar':'egg'}})
{'bla.foo.$unicode.bla': 3, 'bla.bar': "egg"}
```

```
>>> DotCollapsedDict({'bla':{'foo':{unicode:{"bla":3}}, 'bar':'egg'}}, remove_under_type=True)
{'bla.foo':{}, 'bla.bar':unicode}
```

```
>>> dic = {'bar':{'foo':3}, 'bla':{'g':2, 'h':3}}
>>> DotCollapsedDict(dic, reference={'bar.foo':None, 'bla':{'g':None, 'h':None}})
{'bar.foo':3, 'bla':{'g':2, 'h':3}}
```

class mongokit.schema_document.**DotedDict**(*doc=None*, *warning=False*)
Dot notation dictionary access

class mongokit.schema_document.**DotExpandedDict**(*key_to_list_mapping*)
A special dictionary constructor that takes a dictionary in which the keys may contain dots to specify inner dictionaries. It's confusing, but this example should make sense.

```
>>> d = DotExpandedDict({'person.1.firstname': ['Simon'],            'person.1.lastname': ['Willi
>>> d
{'person': {'1': {'lastname': ['Willison'], 'firstname': ['Simon']},
'2': {'lastname': ['Holovaty'], 'firstname': ['Adrian']}}}
>>> d['person']
{'1': {'lastname': ['Willison'], 'firstname': ['Simon']}, '2': {'lastname': ['Holovaty'], 'first
>>> d['person']['1']
{'lastname': ['Willison'], 'firstname': ['Simon']}
```

# Gotcha: Results are unpredictable if the dots are "uneven": >>> DotExpandedDict({'c.1': 2, 'c.2': 3, 'c': 1}) {'c': 1}

exception mongokit.schema_document.**DuplicateDefaultValueError**

exception mongokit.schema_document.**DuplicateRequiredError**

class mongokit.schema_document.**i18n**(*field_type=None*, *field_name=None*)
CustomType to deal with i18n

mongo_type
alias of list

to_bson(*value*)

to_python(*value*)

exception mongokit.schema_document.**i18nError**

exception mongokit.schema_document.**ModifierOperatorError**

exception mongokit.schema_document.**RequireFieldError**

class mongokit.schema_document.**SchemaDocument**(*doc=None*, *gen_skel=True*, *_gen_auth_types=True*, *_validate=True*, *lang='en'*, *fallback_lang='en'*)
A SchemaDocument is dictionary with a building structured schema The validate method will check that the document match the underling structure. A structure must be specify in each SchemaDocument.

```
>>> class TestDoc(SchemaDocument):
...     structure = {
...         "foo":six.text_type,
...         "bar":int,
...         "nested":{
...            "bla":float}}
```

unicode, *int*, *float* are python types listed in *mongokit.authorized_types*.

```
>>> doc = TestDoc()
>>> doc
{'foo': None, 'bar': None, 'nested': {'bla': None}}
```

A SchemaDocument works just like dict:

```
>>> doc['bar'] = 3
>>> doc['foo'] = "test"
```

We can describe fields as required with the required attribute:

```
>>> TestDoc.required_fields = ['bar', 'nested.bla']
>>> doc = TestDoc()
>>> doc['bar'] = 2
```

Validation is made with the *validate()* method:

```
>>> doc.validate()
Traceback (most recent call last):
...
RequireFieldError: nested.bla is required
```

Default values can be set by using the attribute default_values :

```
>>> TestDoc.default_values = {"bar":3, "nested.bla":2.0}
>>> doc = TestDoc()
>>> doc
{'foo': None, 'bar': 3, 'nested': {'bla': 2.0}}
>>> doc.validate()
```

Validators can be added in order to validate some values :

```
>>> TestDoc.validators = {"bar":lambda x: x>0, "nested.bla": lambda x: x<0}
>>> doc = TestDoc()
>>> doc['bar'] = 3
>>> doc['nested']['bla'] = 2.0
>>> doc.validate()
Traceback (most recent call last):
...
ValidationError: nested.bla does not pass the validator <lambda>
```

If you want to use the dot notation (ala json), you must set the *use_dot_notation* attribute to True:

```
>>> class TestDotNotation(SchemaDocument):
...     structure = {
...         "foo":{ "bar":unicode}
...     }
...     use_dot_notation=True
```

```
>>> doc = TestDotNotation()
>>> doc.foo.bar = u"bla"
>>> doc
{"foo":{"bar":u"bla}}
```

authorized_types = [<class 'NoneType'>, <class 'bool'>, <class 'int'>, <class 'float'>, <class 'list'>, <class 'dict'>, <cl

default_values = {}

dot_notation_warning = False

generate_skeleton()

validate and generate the skeleton of the document from the structure (unknown values are set to None)

get_lang()

i18n = []

raise_validation_errors = True

required_fields = []

set_lang(*lang*)

`skip_validation` = False

`structure` = None

`use_dot_notation` = False

`use_schemaless` = False

`validate``()`
validate the document.

This method will verify if :

- the doc follow the structure,
- all required fields are filled

Additionally, this method will process all validators.

`validators` = {}

exception `mongokit.schema_document.`**`SchemaDocumentError`**

class `mongokit.schema_document.`**`SchemaProperties`**

exception `mongokit.schema_document.`**`SchemaTypeError`**

class `mongokit.schema_document.`**`Set`**`(`*structure_type=None*`)`
SET custom type to handle python set() type

`init_type`
alias of `set`

`mongo_type`
alias of `list`

`python_type`
alias of `set`

`to_bson``(`*value*`)`

`to_python``(`*value*`)`

`validate``(`*value*, *path*`)`

exception `mongokit.schema_document.`**`StructureError`**

exception `mongokit.schema_document.`**`ValidationError`**

5.7.15 Versioned Document

class `mongokit.versioned_document.`**`RevisionDocument`**`(`*doc=None*, *gen_skel=True*, *collection=None*, *lang='en'*, *fallback_lang='en'*`)`

`structure` = {'id': <class 'str'>, 'revision': <class 'int'>, 'doc': <class 'dict'>}

class `mongokit.versioned_document.`**`VersionedDocument`**`(`*doc=None*, **args*, ***kwargs*`)`
This object implement a vesionnized mongo document

`delete``(`*versioning=False*, **args*, ***kwargs*`)`
if versioning is True delete revisions documents as well

`get_last_revision_id``()`

`get_revision``(`*revision_number*`)`

get_revisions()

remove(*query*, *versioning=False*, **args*, ***kwargs*)
if versioning is True, remove all revisions documents as well. Be careful when using this method. If your query match tons of documents, this might be very very slow.

save(*versioning=True*, **args*, ***kwargs*)

5.8 Changelog

5.8.1 Version 1.0.0

- Added python 3 support (thanks to @aquavitae)
- Big code clean up (thanks to @Winnetou)
- Updated documentation
- Added read the docs documentation
- Improved documentation structure

5.8.2 Version 0.9.1

- fixed #131 - Use PEP8 recommendation for import
- fixed tests (thanks @JohnBrodie and @bneron)
- Added a Makefile for running tests in venv (thanks to @gulbinas)
- fixed pep8 error (thanks to @gulbinas)
- added support for MongoReplicaSetClient (thanks to @inabhi9)
- Added __*getstate*__ and __*setstate*__ to DotedDict and i18nDotedDict. Problems appeared here when pickling mongokit documents due to apparent lack of these functions. (thanks to @petersng)
- Fixed english mistake and typos into the documentation (thanks to @biow0lf, @SeyZ, @gianpaj and @1123)
- Fixed inherited queries when accessing cursor by index (thanks to @asivokon)
- changed the namespace on schema document errors (thanks to @rtjoseph11)

5.8.3 Version 0.9.0

- now MongoKit required PyMongo >= 2.5
- find_and_modify returns None if the query fails (thanks to @a1gucis)
- Fix off-by-one error on SchemaDocument (thanks to @John Brodie)
- Fix inherited queries (issue #106) (thanks to @effem-git)
- Fix for serialization of nested structures with type validation (thanks to @LK4D4)
- Remove unnecessary path arguments in to_json._convert_to_python (thanks to @Alexandr Morozov)
- big refactorization by using multiple inheritance for DRYness (thanks to @liyanchang)
- Add find_fulltext method for convenience (thanks to @astronouth7303) (not official and not documented yet)

- Allow text indexes in document definitions (thanks to @astronouth7303)
- Adding replica set support (thanks to @liyanchang)
- Fix typos on README (thanks to @girasquid)
- add pagination helper (not yed documented)(thanks to @jarrodb) https://github.com/namlook/mongokit/blob/master/mongokit/pag

5.8.4 Version 0.8.3

- allow keyword arguments (like read_preferences, slave_okay, etc) to be set in Connection (thanks to Peter S Ng)
- Add find_and_modify again. It was removed by an unexpected rollback.
- use MongoClient with MasterSlaveConnection

5.8.5 Version 0.8.2

- fix #101 - validators condition fix
- fix #110 - support PyMongo >= 2.4 (import MongoClient) – thanks to @mattbodman and @zavatskiy
- Fixed some spelling/grammar (thanks to @gekitsuu)

5.8.6 Version 0.8.1

- support pymongo 2.3
- small updates to validation messages (Merge pull request #94 from unpluggd/master)
- Fixed typo when throwing MaxDocumentSizeError in validate() (thanks to Andy Pavlo)
- added fix for unconditional access to __wrap on cursors (thanks to David T. Lehmann)
- Add .travis.yml for Travis CI (http://travis-ci.org/) (Marc Abramowitz)
- Fixed rendering issue in the docs. (thanks to Dougal Matthews)
- tweaked the error messages in validation for missing and unknown fields to aid in debugging projects (thanks to Phillip B Oldham)

5.8.7 Version 0.8.0

- Add spec file for rpm-based distributions (Merge pull request #63 from linuxnow/master)
- change document size limitation for mongodb 1.8 or later. Thanks to Aleksey Sivokon (Merge pull request #74 from key/master)
- validation of "" for an int (Merge pull request #79 from barnybug/master)
- Fix exception when loading documents with a custom type field missing (Merge pull request #80 from barnybug/master)
- Big documentation restructuration made by Sean Lynch (Merge pull request #82 from sean-lynch/master)
- Using rename no longer causes migrations throw an exception (Merge pull request #86 from matthewh/master)
- Some test is modified and added tox (Merge pull request #91 from aircastle/modifiytest)
- Replace pymongo.objectid with bson.objectid (Merge pull request #88 from behackett/master)

- Added Support for additional keyword-arguments for index-creation (Merge pull request #85 from mfelsche/master)
- Remove anyjson dependency and use builtin json instead

Thank you all for all your patches !

5.8.8 Version 0.7.2

- add inherited queries support (please see http://github.com/namlook/mongokit/wiki/Inherited-queries for more details)

5.8.9 Version 0.7.1

- change MongokitMasterSlaveConnection to MasterSlaveConnection for consistency
- fix #57 – support pymongo > 1.9 in grid.py
- fix #45 – remove automatic index creation
- fix #43 – slicing a cursor should return a mongokit document, not dict
- Don't try to convert None struct to json (patch from @mLewisLogic thanks !)
- fix schemaless issue (thanks to Mihai Pocorschi for reporting it)

5.8.10 Version 0.7

- add *use_schemaless* feature ! please see the documentation for more information
- **Add equality test for mongokit operators (thanks to @allancaffee)** This allows developers to write unit tests on the structure of their document classes when operators are used
- roll back find_and_modify for master branch (need pymongo 1.10 for that)
- many documentation fixes
- fix #55 – Bug in VersionedDocument remove() method
- fix #53 – Fixed a few spelling errors in README
- fix #52 – Advanced bulk migration docs example is broken
- fix #51 – pymongo.dbref is deprecated, use bson.dbref instead
- fix #49 – Can't specify default values for lists of embedded objects
- fix #48 – uuid.UUID support
- fix #41 – add basestring to authorized types
- fix #40 – Made some enhancements
- fix #39 – KeyError when saving partially loaded documents
- fix #34 – add find_and_modify method to Document
- fix #32 – allow the structure to be empty (was: document.to_json())
- fix #24 – Don't handle __database__/__collection__ attribute for virtual documents

5.8.11 Version 0.6

- fix error when check is True. Thanks to @dasmith for the patch
- Many english corrections in the documentation thanks to @allancaffee
- spliting doc and refactoring documentation
- remove unused MongoDocumentCursor

5.8.12 Version 0.5.13.1

- fix #26 – unable to install (debian lenny, py 2.5)
- fix #25 – put the new url into the setup.py

5.8.13 Version 0.5.13

- fix #21 – required_fields weird behavior with autorefs
- fix #19 – 'checked' field not listed in 'indexes' section
- fix #20 – creating index on fields not in structure + optimize index generation
- fix #18 – typo in the doc
- fix import. Dbref isn't in pymongo package anymore
- fix deprecation warning from pymongo's from_dict
- fix #8 – allow to access Document via the db

5.8.14 Version 0.5.12.1

- fix #17 – got an unexpected keyword argument 'from_son'
- fix #15 – typo in the doc

5.8.15 Version 0.5.12

- allow register method to be a decorator (thanks to Christopher Grebs for the inspiration)
- get ride of MongoDocumentCursor and use a subclass of pymongo's Cursor instead
- structure and descriptors validation is now done at object creation (not instantiation)
  - *advantage* : mongokit is 40% faster
  - *beware* : if you put a Document into a structure for reference, mongokit doesn't check anymore if use_autorefs is set
- add i18n descriptor validation + better i18n support
- code cleaning

5.8.16 Version 0.5.11

- support latest pymongo version
- some changes in GridFS support (please read http://namlook.github.com/mongokit/gridfs.html)
- Deprecate atomic_save feature
- remove libmagic import from grid.py : to many trouble with this lib, we have to find another way to guess the content-type
- fix #79 – Tries to migrate non-saved document
- fix #70 – Set changes from set to list when a validation error occurs
- add contributor + fix email address to prevent spam
- fix deprecation warning of Python 2.6
- fix issue with validation and migration
- fix #75 – add "version" attribute to module

5.8.17 Version 0.5.10

- fix bug in autorefs when reference in double list

5.8.18 Version 0.5.9

- minors fixes

5.8.19 Version 0.5.8

- add rewind to cursor
- killed many bug in *from_json()*
- fix #66 - local variable 'l_objs' referenced before assignment
- fix #61 - Issue with indexing on multi-columns with directions

5.8.20 Version 0.5.7

- fix #63 - Creating index for each document instance operation. Brings speed improvements
- fix #60 - autorefs doesn't work with complex structures
- fix #62 - Dereference to model. Thanks to Christian Joudrey for the patch
- fix #64 - error with atomic_save when using embed document
- fix #65 - Lazy migrations with dict in list and documentation fix

5.8.21 Version 0.5.6

- add atomic update (just save the document again)
- add init_type to CustomType. This allow to fill an empty skeleton at instanciation
- add debian package build rules. Thanks to Sebastien Estienne
- add lazy migration and bulk migration support
- fix a bug in CustomType
- add 'check' option in indexes descriptor
- add untyped field support
- fix #58 - Document Validators not working for CustomType
- improve DotCollapsedDict by adding reference structure

5.8.22 Version 0.5.5

- fix 54 - Add reload method. Please read the documentation
- put generate_index into Document.__init__. This is useful for instanciating Document like this : MyDoc(collection=mycol)
- fix #44 - add set type support + add validate() method to CustomType
- fix #52 - Custom validation error messages (thanks to @cjoudrey for the patch)
- fix #50 - need optimizations in connection (won 20s on the benchmark)
- fix #48 - Tuple assignment does not convert to list
- fix 49 - KeyError when using deep nested autorefs

5.8.23 Version 0.5.4

- A lot of features in GridFS with api change
- fix bug in autorefs
- fix #37 - find_random crash if no collection is empty
- fix #38 - OverflowError in doc.to_json_type() when used over the datetime 2038
- fix #41 - Warnings when setting attributes before enabling use_dot_notation
- fix #40 - Better exception on bad structure. Thanks to peterbe for the patch
- fix #43 - Add ability to collect errors in one place instead of throwing exceptions while validating
- add *_dot_notation_warning* attribute. If false, disable all dot notation related warning
- add patch to enable data load from map/reduce. See http://groups.google.com/group/mongokit/msg/34efea4c178573d7
- fix bug spotted by Sebastien Estienne - error when using skip_validation with required_fields. Thanks
- fix issue while using {unicode:unicode} in structure and i18n at the same time

5.8.24 Version 0.5.3

- fix default_value issue when using with dict and list (see #35)
- fix bug reported by Andrew Degtiariov : http://bit.ly/c1vcUv
- add clone and explain method to MongoDocumentCursor
- add distinct to cursor (thanks to Flaper87)
- fix index test
- fix : when a field is added to a saved document and not specified in the structure, the validation wasn't work properly
- use current database if DBRef has no database information. Please, see the doc
- support of pymongo 1.4

5.8.25 Version 0.5.2

- bugs fix in json import/export
- bugs fix in default values and required values
- gridfs support

5.8.26 Version 0.5.1

- *save()* doesn't return `self` anymore (was an API monster)
- fix bug in *find_one()* method. Now returns None if no Document is found
- fix bug when using default values
- adding i18n list support
- add i18n inheritance support
- adding index inheritance support

5.8.27 Version 0.5

- refactoring API which is getting much much more cleaner. Please see the migration page to keep your code up to date
- 100% code coverage by 162 unit tests
- lot of bug fix (too many to list them here)
- add document size validation
- add cross database reference support
- i18n support

5.8.28 Version 0.4

- add autoref support to belong_to (delete cascade)
- changing collection dynamically
- add immutable field (python tuple support)
- add direction and ttl to index support
- add connection sharing support
- add json import/export for MongoDocument
- full relation support (related_to)
- add long type support

5.8.29 Version 0.3.3

- add autoref support (thanks to @bwmcadams)
- add mongodb index support (thanks to @marcammann)
- adding CustomType (original idea from Phillip Oldham)
- support now all type of subclassed supported type
- add "delete cascade" feature
- add the possibility to skip the validation layer for more performances
- fix issue while passing queries to fetch() and update tutorial
- self._collection must not be None in __init__
- fix #11 - pylons_env extension documentation typo
- add more complete test + docstring
- fix issue #9 - bug with custom_types and nested dict in list

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index