Django MongoDB Engine

Release

Sep 26, 2017

Contents

Django MongoDB Engine is a MongoDB backend for Django, the Python Web framework for perfectionists with deadlines.

This documentation is split into several sections:

- *The tutorial* introduces you to non-relational schema design and tools provided by Django MongoDB Engine to put that design into action.
- Our *topical guides* explain every feature of Django MongoDB Engine in great detail – they'll provide you with usage guides, API overviews, code samples etc.
- The *API reference* has all the lower-level details on functions, methods, classes, settings, modules and whatnot.
- In the *Meta section*, we collect information about the project itself: *What has changed between versions*, how development happens and *how you can get involved*.

CHAPTER 1

Contents

Tutorial

The goal of this step-by-step tutorial is to introduce you to non-relational schema design and the tools Django MongoDB Engine provides to put that design into action.

This tutorial assumes that you are already familiar with Django and have a basic idea of MongoDB operation and a *configured MongoDB installation*.

Our example project covers the development of a simple single-user blog application with tag and comment capabilities.

Note: When you try out the shell examples given in this tutorial (which you should!) it is important to remember that model changes will have no effect until you restart the shell.

Non-Relational Schema Design

If you come from a relational background a non-relational database may seem counter-intuitive or completely odd since most non-relational databases are document and multi-key oriented and provide a different method of querying and excluding data.

Perhaps a good way to get used to non-relational style data modeling is to ask yourself "What would I never do in SQL".

Because most relational databases lack proper list data structures you may typically model the Posts Tags Comments relationship using three models/tables, one table per entity type.

Organizing your data using multiple relationships is the exact opposite of what we will do for our non-relational data model: Have one single collection (table) for everything and store tags and comments in simple lists.

Here is a simple model for blog posts, designed for non-relational databases:

```
from django.db import models

from djangotoolbox.fields import ListField


class Post(models.Model):
    title = models.CharField()
    text = models.TextField()
    tags = ListField()
    comments = ListField()
```

Let's try this out. Fire up a Django shell and add a post:

```
>>> from nonrelblog.models import Post
>>> post = Post.objects.create(
...     title='Hello MongoDB!',
...     text='Just wanted to drop a note from Django. Cya!',
...     tags=['mongodb', 'django']
... )

Surely we want to add some comments.

>>> post.comments
[]
>>> post.comments.extend(['Great post!', 'Please, do more of these!'])
>>> post.save()

Look and see, it has actually been saved!

>>> Post.objects.get().comments
[u'Great post!', u'Please, do more of these!']
```

In the MongoDB shell, you can see how the resulting data record looks like:

```
{
  "_id" : ObjectId("..."),
  "tags" : ["mongodb", "django"],
  "text" : "Just wanted to drop a note from Django. Cya!",
  "title" : "Hello MongoDB!",
  "comments" : [
    "Great post!",
    "Please, do more of these!"
  ]
}
```

You may have noticed something's missing from the `Post` class: We have no information about the date and time our posts are created! Fixed easily.

Migration-Free Model Changes

Happily, because MongoDB is schema-less, we can add new fields to our model without corrupting existing data records ("documents"). Forget about migrations!

So, adding a new field boils down to... adding a new field.

```
class Post(models.Model):
    created_on = models.DateTimeField(auto_now_add=True, null=True) # <---
```

```
    title = models.CharField(max_length=255)
    text = models.TextField()
    tags = ListField()
    comments = ListField()
```

One thing to keep in mind is what happens to our old posts: Because they miss a `created_on` value, when fetching them in Django, the `created_on` attribute will be set to the `DateTimeField` default value, *None*. To allow *None* as value, we have to pass `null=True`.

```
We can use database records even though they were created
with an older version of or model schema:

>>> from nonrelblog.models import Post
>>> old_post = Post.objects.all()[0]
>>> old_post.created_on is None
True
>>> new_post = Post.objects.create()
>>> new_post.created_on is None
False
```

There's another flaw in our design: We can't store any comment meta information like author name/email and creation time. We'll tackle that in the next section.

Embedded Models

So far, we used to store comments as a list of strings. We'll have to rework that design in order to store additional information for each comment.

Let's first design our model for comments.

```
class Comment(models.Model):
    created_on = models.DateTimeField(auto_now_add=True)
    author_name = models.CharField(max_length=255)
    author_email = models.EmailField()
    text = models.TextField()
```

The BSON representation of this model looks like this:

```
{
  'created_on': ISODate('...'),
  'author_name': 'Bob',
  'author_email': 'bob@example.org',
  'text': 'The cake is a lie'
}
```

MongoDB allows to have objects within objects – called "subobjects" or "embedded objects" – so we could also represent this as follows:

```
{
  'created_on': ISODate('...'),
  'author' : {
    'name': 'Bob',
    'email': 'bob@example.org'
  },
  'text' : 'The cake is a lie'
}
```

Django itself does not allow such nesting – because there's no such thing in SQL – but Django MongoDB Engine provides the tools to do anyway.

To embed instances of models into other models, we can use *EmbeddedModelField*:

```
from djangotoolbox.fields import EmbeddedModelField
    tags = ListField()
    comments = ListField(EmbeddedModelField('Comment')) # <---


class Comment(models.Model):
    created_on = models.DateTimeField(auto_now_add=True)
    author = EmbeddedModelField('Author')
    text = models.TextField()


class Author(models.Model):
    name = models.CharField()
```

Let's hop into the Django shell and test this:

```
>>> from nonrelblog.models import Comment, Author
>>> Comment(
...     author=Author(name='Bob', email='bob@example.org'),
...     text='The cake is a lie'
... ).save()
>>> comment = Comment.objects.get()
>>> comment.author
<Author: Bob (bob@example.org)>
```

In the same way, we can embed `Comment` objects into the `comments` list of a blog post, by *combining ListField and EmbeddedModelField*:

```
class Post(models.Model):
    created_on = models.DateTimeField(auto_now_add=True, null=True)
    title = models.CharField()
    text = models.TextField()
    tags = ListField()
    comments = ListField(EmbeddedModelField('Comment')) # <---
```

We should mess around with our new `Post` model at this point.

```
>>> Post(
...     title='I like cake',
...     comments=[comment]
... ).save()
>>> post = Post.objects.get(title='I like cake')
>>> post.comments
[<Comment: Comment object>]
>>> post.comments[0].author.email
u'bob@example.org'
```

Here's how this post is represented in MongoDB:

```
{
  "_id" : ObjectId("..."),
  "tags" : [ ],
  "text" : "",
  "title" : "I like cake",
```

```
  "comments" : [
    {
      "text" : "The cake is a lie",
      "created_on" : ISODate("..."),
      "id" : null,
      "author" : {
        "email" : "bob@example.org",
        "name" : "Bob",
        "id" : null
      }
    }
  ],
  "created_on" : ISODate("...")
}
```

Neat, isn't it?

Using lists, dictionaries and embedded objects, you can design your database schema very similar to the structure of the Python objects used all over your code. No need to squeeze your objects into primitive non-relational schemas.

Adding Some Templates

To make our app actually useful, it's time to add some views. Here's how your post overview page could look like:

```
<h1>Post Overview</h1>
{% for post in post_list %}
  <h2><a href="{% url post_detail post.id %}">{{ post.title }}</a></h2>
  <p>
    {{ post.created_on }} |
    {{ post.comments|length }} comments |
    tagged {{ post.tags|join:', ' }}
  </p>
{% endfor %}
```

Pretty straightforward. Here's the single post template:

```
<h1>{{ post.title }}</h1>
<p>{{ post.created_on }}</p>
<p>{{ post.text }}<p>
<h2>Comments</h2>
{% for comment in post.comments %}
  <h3>{{ comment.author.name }} <small>on {{ comment.created_on }}</small></h3>
  {{ comment.text }}
{% endfor %}
```

By using Django's Generic Views, we even don't have to write any views, so all that's left is mapping URLs to those templates:

```
from django.conf.urls.defaults import patterns, url
from django.views.generic import ListView, DetailView

from models import Post


post_detail = DetailView.as_view(model=Post)
post_list = ListView.as_view(model=Post)
```

```
urlpatterns = patterns('',
    url(r'^post/(?P<pk>[a-z\d]+)/$', post_detail, name='post_detail'),
    url(r'^$', post_list, name='post_list'),
)
```

A fully working example project can be found in `docs/source/code/tutorial/v3/`.

Uploading Files to GridFS

To make our blog less boring, we should add some nice pictures.

As MongoDB disciples, what comes to mind when thinking about storing files? Of course! GridFS!

Django MongoDB Engine provides a *Django storage backend for GridFS* that allows you to use GridFS like any other file storage:

```
from django_mongodb_engine.storage import GridFSStorage


gridfs_storage = GridFSStorage()
```

```
from django.db import models

from gridfsuploads import gridfs_storage


class FileUpload(models.Model):
    created_on = models.DateTimeField(auto_now_add=True)
    file = models.FileField(storage=gridfs_storage, upload_to='/')
```

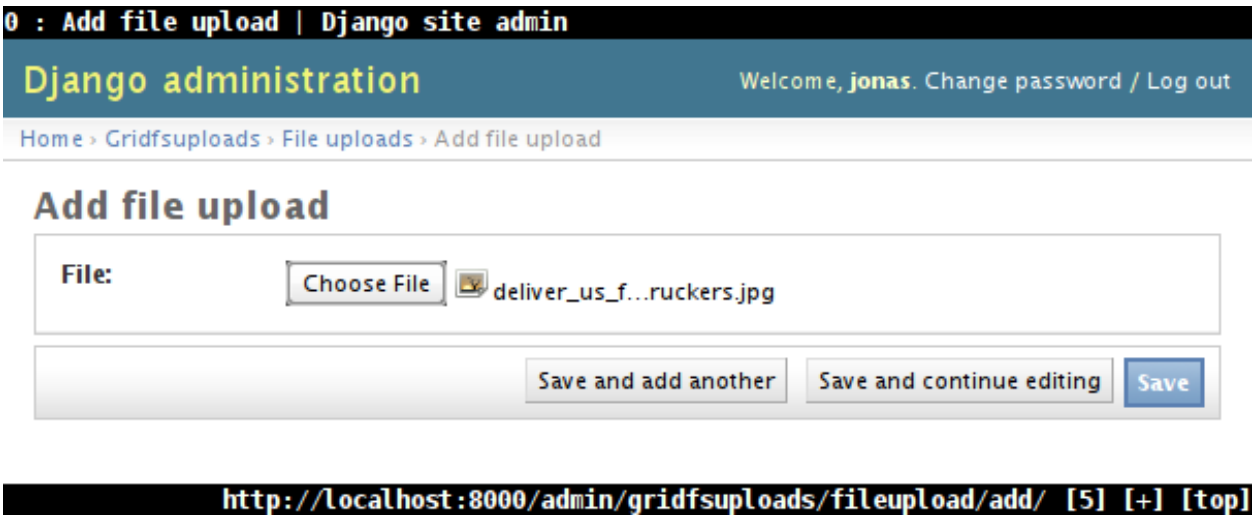
We can now use the Django admin to upload files to GridFS.

Next step is to write a serve view:

```
from mimetypes import guess_type

from django.conf import settings
from django.http import HttpResponse, Http404

from gridfs.errors import NoFile
from gridfsuploads import gridfs_storage
from gridfsuploads.models import FileUpload


if settings.DEBUG:

    def serve_from_gridfs(request, path):
        # Serving GridFS files through Django is inefficient and
        # insecure. NEVER USE IN PRODUCTION!
        try:
            gridfile = gridfs_storage.open(path)
        except NoFile:
            raise Http404
        else:
            return HttpResponse(gridfile, mimetype=guess_type(path)[0])
```

Warning: Serving files through such a view is *inefficient* and *insecure*. **Never** use this in production! There are much superior tools for serving files out of GridFS, e.g. nginx-gridfs.

We can now upload files via the admin...



... and retrieve them under `/uploads/`:

As always, the full project source can be found in `docs/source/code/tutorial/v4/`.

Using Map/Reduce

Our last quest is to count the number of comments each author has made.

This could be done in plain Django but would be very inefficient because we would have to literally fetch all posts (with all comments) from the database.

Instead, we're going to use *Map/Reduce* to accomplish the task.

Programmer's introduction to Map/Reduce: The *map* function gets called for each document and emits one or more *key-value pairs*. The *reduce* function is passed a *key* and a list of *values* and reduces them to a single resulting value. The result of such a Map/Reduce operation is a list of key-value pairs, the keys being those emitted by the map function and the values those resulting from the reduce function.

Our map function emits a `(author, 1)` pair for each comment.

```
function map() {
  /* `this` refers to the current document */
  this.comments.forEach(function(comment) {
    emit(comment.author.name, 1);
  });
}
```

The reduce function sums up all the ones emitted by the map function.

```
function reduce(id, values) {
  /* [1, 1, ..., 1].length is the same as sum([1, 1, ..., 1]) */
  return values.length;
}
```

Map/Reduce support is added to Django's ORM using a custom `Manager` which is installed to the `Post` model as follows:

```
from django_mongodb_engine.contrib import MongoDBManager
...
class Post(models.Model):
   ...
   objects = MongoDBManager()
```

Ready to Map/Reduce?

```
>>> from nonrelblog.models import *

Add some data so we can actually mapreduce anything.
Bob:   3 comments
Ann:   6 comments
Alice: 9 comments
>>> authors = [Author(name='Bob', email='bob@example.org'),
...            Author(name='Ann', email='ann@example.org'),
...            Author(name='Alice', email='alice@example.org')]
>>> for distribution in [(0, 1, 2), (1, 2, 3), (2, 3, 4)]:
...     comments = []
...     for author, ncomments in zip(authors, distribution):
...         comments.extend([Comment(author=author)
...                          for i in xrange(ncomments)])
...     Post(comments=comments).save()

------------------------
Kick off the Map/Reduce:
------------------------
>>> pairs = Post.objects.map_reduce(mapfunc, reducefunc, out='temp',
...                                 delete_collection=True)
>>> for pair in pairs:
...     print pair.key, pair.value
Alice 9.0
Ann 6.0
Bob 3.0
```

This is worth a review.

- `MongoDBManager.map_reduce()` returns an iterator yielding `MapReduceResult` objects.
- The third argument to `map_reduce()` is the name of the collection where the Map/Reduce results should go to.

- The fourth argument, *delete_collection*, tells Django MongoDB Engine to delete the temporary collection passed as third argument after the Map/Reduce result iterator is exhausted.
- The resulting counts are floats because Javascript does not distinguish between integers and floating point numbers.

Lastly, a quick word of warning. Map/Reduce is designed to be used for *one-time operations* – although it performs very well, it's definitely not something you would want to execute on a per-request basis. *Don't use Map/Reduce in "hot" parts of your code*.

Where to go from here

This tutorial should've given you an idea about how easy it is to combine Django and MongoDB using Django MongoDB Engine to produce simple, scalable applications.

Hopefully you've learned something useful for your next Django project that you should begin hacking on *now*. Go build something cool, and *let us know about it*!

You can always come back to this documentation as you need to learn new tricks:

- Our *topical guides* explain every feature of Django MongoDB Engine in great detail – they'll provide you with usage guides, API overviews, code samples etc.
- The *API reference* has all the lower-level details on functions, methods, classes, settings, modules and whatnot.
- In the *Meta section*, we collect information about the project itself: *What has changed between versions*, how development happens and *how you can get involved*.

If you need support, don't hesitate to write to our *mailing list*.

Also, we'd love to see you *getting involved in Django MongoDB Engine's development*!

- Fix the documentation. None of the Django MongoDB Engine developers are native English speakers, so this docs are probably full of typos and weird, ungrammatical or incomprehensible phrasings. Every typo is worth reporting!
- Extend and improve the documentation. We appreciate any contribution!
- Blog/write about Django MongoDB Engine, and *send us a link* to your work.
- *Report bugs and feature requests*.
- Finally, *send pull requests or patches* containing bug fixes, new features and code improvements.

Setup

This page explains how to install and configure a Django/MongoDB setup.

Installation

Django MongoDB Engine depends on

- Django-nonrel, a fork of Django that adds support for non-relational databases
- djangotoolbox, a bunch of utilities for non-relational Django applications and backends

It's highly recommended (although not required) to use a virtualenv for your project to not mess up other Django setups.

virtualenv

If not already installed, grab a copy from the Cheeseshop:

```
pip install virtualenv
```

To set up a virtual environment for your project, use

```
virtualenv myproject
```

To join the environment, use (in Bash):

```
source myproject/bin/activate
```

Django-nonrel

```
pip install git+https://github.com/django-nonrel/django@nonrel-1.5
```

djangotoolbox

```
pip install git+https://github.com/django-nonrel/djangotoolbox
```

Django MongoDB Engine

You should use the latest Git revision.

```
pip install git+https://github.com/django-nonrel/mongodb-engine
```

Configuration

Database setup is easy (see also the Django database setup docs):

```
DATABASES = {
   'default' : {
      'ENGINE' : 'django_mongodb_engine',
      'NAME' : 'my_database'
   }
}
```

Django MongoDB Engine also takes into account the `HOST`, `PORT`, `USER`, `PASSWORD` and `OPTIONS` settings.

Possible values of `OPTIONS` are described in the *settings reference*.

Done!

That's it! You can now go straight ahead developing your Django application as you would do with any other database.

Topics

Lists & Dicts

Django MongoDB Engine provides two fields for storing arbitrary (BSON-compatible) Python `list` and `dict` objects in Django model objects, *ListField* and *DictField*, which can be used to store information that is not worth a separate model or that should be queryable in efficient manner (using an index).

Both fields may optionally be provided with type information. That restricts their usage to one single type but has the advantage of automatic type checks and conversions.

ListField

Stores Python lists (or any other iterable), represented in BSON as arrays.

```
from djangotoolbox.fields import ListField

class Post(models.Model):
    ...
    tags = ListField()
```

```
>>> Post(tags=['django', 'mongodb'], ...).save()
>>> Post.objects.get(...).tags
['django', 'mongodb']
```

The typed variant automatically does type conversions according to the given type:

```
class Post(models.Model):
    ...
    edited_on = ListField(models.DateTimeField())
```

```
>>> post = Post(edited_on=['1010-10-10 10:10:10'])
>>> post.save()
>>> Post.objects.get(...).edited_on
[datetime.datetime([1010, 10, 10, 10, 10, 10])]
```

As described *in the tutorial*, ListFields are very useful when used together with *Embedded Models* to store lists of sub-entities to model 1-to-n relationships:

```
from djangotoolbox.fields import EmbeddedModelField, ListField

class Post(models.Model):
    ...
    comments = ListField(EmbeddedModelField('Comment'))

class Comment(models.Model):
    ...
    text = models.TextField()
```

Please head over to the *Embedded Models* topic for more about embedded models.

SetField

Much like a *ListField* except that it's represented as a `set` on Python side (but stored as a list on MongoDB due to the lack of a separate set type in BSON).

DictField

Stores Python dicts (or any dict-like iterable), represented in BSON as subobjects.

```
from djangotoolbox.fields import DictField

class Image(models.Model):
    ...
    exif = DictField()
```

```
>>> Image(exif=get_exif_data(...), ...).save()
>>> Image.objects.get(...).exif
{u'camera_model' : 'Spamcams 4242', 'exposure_time' : 0.3, ...}
```

The typed variant automatically does type conversion on values. (Not on keys as the are required to be strings on MongoDB.)

```
class Poll(models.Model):
    ...
    votes = DictField(models.IntegerField())
```

```
>>> Poll(votes={'bob' : 3.14, 'alice' : '42'}, ...).save()
>>> Poll.objects.get(...).votes
{u'bob' : 3, u'alice' : 42}
```

DictFields are useful mainly for storing objects of varying shape, i.e. objects whose structure is unknow at coding time. If all your objects have the same structure, you should consider using *Embedded Models*.

Embedded Models

Django MongoDB Engine supports MongoDB's subobjects which can be used to embed an object into another.

Using *ListField* and *DictField* it's already possible to embed objects (`dicts`) of arbitrary shape.

However, EmbeddedModelField (described beneath) is a much more comfortable tool for many use cases, ensuring the data you store actually matches the structure and types you want it to be in.

The Basics

Let's consider this example:

```
from djangotoolbox.fields import EmbeddedModelField

class Customer(models.Model):
    name = models.CharField(...)
    address = EmbeddedModelField('Address')
    ...

class Address(models.Model):
    ...
    city = models.CharField(...)
```

The API feels very natural and is similar to that of Django's relation fields.

```
>>> Customer(name='Bob', address=Address(city='New York', ...), ...).save()
>>> bob = Customer.objects.get(...)
>>> bob.address
<Address: Address object>
>>> bob.address.city
'New York'
```

Represented in BSON, Bob's structure looks like this:

```
{
  "_id": ObjectId(...),
  "name": "Bob",
  "address": {
    ...
    "city": "New York"
  },
  ...
}
```

While such "flat" embedding is useful if you want to bundle multiple related fields into one common namespace – for instance, in the example above we bundled all information about a customers' address into the *address* namespace – there's a much more common usecase for embedded objects: one-to-many relations.

Lists of Subobjects (One-to-Many Relations)

Often, lists of subobjects are superior to relations (in terms of simplicity and performance) for modeling one-to-many relationships between models.

Consider this elegant way to implement the Post Comments relationship:

```
from djangotoolbox.fields import ListField, EmbeddedModelField

class Post(models.Model):
    ...
    comments = ListField(EmbeddedModelField('Comment'))

class Comment(models.Model):
    text = models.TextField()
```

Embedded objects are represented as subobjects on MongoDB:

```
>>> comments = [Comment(text='foo'), Comment(text='bar')]
>>> Post(comments=comments, ...).save()
>>> Post.objects.get(...).comments
[<Comment: Comment object>, <Comment: Comment object>]
```

```
{
  "_id": ObjectId(...),
  ...
  "comments" : [
    {"text": "foo", },
    {"text": "bar"}
  ]
}
```

Generic Embedding

Similar to Django's generic relations, it's possible to embed objects of any type (sometimes referred to as "polymorphic" relationships). This works by adding the model's name and module to each subobject, accompanying the actual data with type information:

```
{
  "_id" : ObjectId(...),
  "stuff" : [
    {"foo" : 42, "_module" : "demoapp.models", "_model" : "FooModel"},
    {"bar" : "spam", "_module" : "demoapp.models", "_model" : "FooModel"}
  ]
}
```

As you can see, generic embedded models add a lot of overhead that bloats up your data records. If you want to use them anyway, here's how you'd do it:

```
class Container(models.Model):
    stuff = ListField(EmbeddedModelField())

class FooModel(models.Model):
    foo = models.IntegerField()

class BarModel(models.Model):
    bar = models.CharField(max_length=255)
```

```
Container.objects.create(
    stuff=[FooModel(foo=42), BarModel(bar='spam')]
)
```

Atomic Updates

Django's support for updates (using the `update()` method) can be used to run atomic updates against a single or multiple documents:

```
Post.objects.filter(...).update(title='Everything is the same')
```

results in a `update()` query that uses the atomic `$set` operator to update the *title* field:

```
.update(..., {'$set': {'title': 'Everything is the same'}})
```

It's also possible to use F() objects which are translated into `$inc` operations. For example,

```
Post.objects.filter(...).update(visits=F('visits')+1)
```

is translated to:

```
.update(..., {'$inc': {'visits': 1}})
```

GridFS

MongoDB's built-in distributed file system, GridFS, can be used in Django applications in two different ways.

In most cases, you should use the GridFS *storage backend* provided by Django MongoDB Engine.

Storage

`GridFSStorage` is a Django storage that stores files in GridFS. That means it can be used with whatever component makes use of storages – most importantly, `FileField`.

It uses a special collection for storing files, by default named "storage".

```
from django_mongodb_engine.storage import GridFSStorage

gridfs = GridFSStorage()
uploads = GridFSStorage(location='/uploads')
```

Warning: To serve files out of GridFS, use tools like nginx-gridfs. **Never** serve files through Django in production!

Model Field

(You should probably be using the *GridFS storage backend*.)

Use `GridFSField` to store "nameless" blobs besides documents that would normally go into the document itself.

All that's kept in the document is a reference (an ObjectId) to the GridFS blobs which are retrieved on demand.

Assuming you want to store a 10MiB blob "in" each document, this is what you *shouldn't* do:

```
# DON'T DO THIS
class Bad(models.Model):
   blob = models.TextField()

# NEITHER THIS
class EventWorse(models.Model):
    blob = models.CharField(max_length=10*1024*1024)
```

Instead, use `GridFSField`:

```
class Better(models.Model):
    blob = GridFSField()
```

A GridFSField may be fed with anything that PyMongo can handle, that is, (preferably) file-like objects and strings.

You'll always get a `GridOut` for documents from the database.

```
>>> doc = Better()

GridFSField takes file-likes (and strings)...
>>> doc.blob = file_like
>>> doc.save()

 ... and always returns GridOuts.
>>> samedoc = Better.objects.get(...)
>>> samedoc.blob
<GridOut object at 0xfoobar>
```

Map/Reduce

Map/Reduce, originally invented at Google, is a simple but powerful technology to efficiently process big amounts of data in parallel.

For this, your processing logic must be split into two phases, the *map* and the *reduce* phase.

The *map phase* takes all the input you'd like to process (in terms of MongoDB, this input are your *documents*) and emits one or more *key-value pairs* for each data record (it "maps" records to key-value pairs).

The *reduce phase* "reduces" that set of key-value pairs into a single value.

This document explains how to use MongoDB's Map/Reduce functionality with Django models.

> **Warning:** MongoDB's Map/Reduce is designed for one-time operations, i.e. it's *not* intended to be used in code that is executed on a regular basis (views, business logic, ...).

How to Use It

Map/Reduce support for Django models is provided through Django MongoDB Engine's `custom Manager` (`What is a manager?`).

```
from django_mongodb_engine.contrib import MongoDBManager

class MapReduceableModel(models.Model):
    ...
    objects = MongoDBManager()
```

The `MongoDBManager` provides a `map_reduce()` method that has the same API as PyMongo's `map_reduce()` method (with the one exception that it adds a *drop_collection* option).

```
>>> MapReduceableModel.objects.map_reduce(mapfunc, reducefunc, output_collection, ...)
```

For very small result sets, you can also use in-memory Map/Reduce:

```
>>> MapReducableModel.objects.inline_map_reduce(mapfunc, reducefunc, ...)
```

It's also possible to run Map/Reduce against a subset of documents in the database:

```
>>> MapReduceableModel.objects.filter(...).map_reduce(...)
```

Both the map and the reduce function are written in Javascript.

`map_reduce()` returns an iterator yielding `MapReduceResult` objects.

Special Reduce Function Rules

A sane reduce function must be both associative and commutative – that is, in terms of MongoDB, the following conditions must hold true:

```
# Value order does not matter:
reduce(k, [A, B]) == reduce(k, [B, A])
# Values may itself be results of other reduce operations:
reduce(k, [reduce(k, ...)]) == reduce(k, ...)
```

This is because in order to be able to process in parallel, the reduce phase is split into several sub-phases, reducing parts of the map output and eventually merging them together into one grand total.

Example

(See also *the example in the tutorial* and Wikipedia, from which I stole the idea for the example beneath.)

As an example, we'll count the number of occurrences of each word in a bunch of articles. Our models could look somewhat like this:

```
from django_mongodb_engine.contrib import MongoDBManager


class Article(models.Model):
    author = models.ForeignKey('Author')
    text = models.TextField()

    objects = MongoDBManager()
```

Our map function emits a `(word, 1)` pair for each word in an article's text (In the map function, *this* always refers to the current document).

```
function() {
  this.text.split(' ').forEach(
    function(word) { emit(word, 1) }
  )
}
```

For an input text of "Django is named after Django Reinhardt", this would emit the following key-value pairs:

```
Django : 1
is : 1
named : 1
after : 1
Django : 1
Reinhardt : 1
```

This pairs are now combined in such way that no key duplicates are left.

```
is : [1]
named : [1]
after : [1]
Django : [1, 1]
Reinhardt : [1]
```

To further process these pairs, we let our reduce function sum up all occurrences of each word

```
function reduce(key, values) {
  return values.length; /* == sum(values) */
}
```

so that the final result is a list of key-"sum"-pairs:

```
is : 1
named : 1
after : 1
Django : 2
Reinhardt : 1
```

Show Me the Codes

Here's a full example, using the models and functions described above, on how to use Django MongoDB Engine's Map/Reduce API.

```
from django.db import models

from django_mongodb_engine.contrib import MongoDBManager


class Article(models.Model):
    author = models.ForeignKey('Author')
    text = models.TextField()

    objects = MongoDBManager()


class Author(models.Model):
    pass
```

```
mapfunc = """
function() {
  this.text.split(' ').forEach(
    function(word) { emit(word, 1) }
  )
}
"""

reducefunc = """
function reduce(key, values) {
  return values.length; /* == sum(values) */
}
"""
```

```
>>> from models import Author, Article

>>> bob = Author.objects.create()
>>> ann = Author.objects.create()

>>> bobs_article = Article.objects.create(author=bob, text="A B C")
>>> anns_article = Article.objects.create(author=ann, text="A B C D E")

Map/Reduce over all articles:
>>> for pair in Article.objects.map_reduce(mapfunc, reducefunc, 'wordcount'):
...     print pair.key, pair.value
A 2.0
B 2.0
C 2.0
D 1.0
```

```
E 1.0

Map/Reduce over Bob's articles:
>>> for pair in Article.objects.filter(author=bob).map_reduce(
            mapfunc, reducefunc, 'wordcount'):
...     print pair.key, pair.value
A 1.0
B 1.0
C 1.0
```

Caching

Note: This document assumes that you're already familiar with Django's caching framework (database caching in particular).

Django MongoDB Cache is a Django database cache backend similar to the one built into Django (which only works with SQL databases).

Cache entries are structured like this:

```
{
  "_id" : <your key>,
  "v" : <your value>,
  "e" : <expiration timestamp>
}
```

Thanks to MongoDB's `_id` lookups being very fast, MongoDB caching may be used as a drop-in replacement for "real" cache systems such as Memcached in many cases. (Memcached is still way faster and does a better caching job in general, but the performance you get out of MongoDB should be enough for most mid-sized Web sites.)

Installation

```
git clone https://github.com/django-nonrel/mongodb-cache
cd mongodb-cache
python setup.py install
```

Setup

Please follow the instructions in the Django db cache setup docs for details on how to configure a database cache. Skip the `createcachetable` step since there's no need to create databases in MongoDB. Also, instead of the default db cache backend name, use `"django_mongodb_cache.MongoDBCache"` as `BACKEND`:

```
CACHES = {
    'default' : {
        'BACKEND' : 'django_mongodb_cache.MongoDBCache',
        'LOCATION' : 'my_cache_collection'
    }
}
```

Django MongoDB Cache will also honor all optional settings the default database cache backend takes care of (`TIMEOUT`, `OPTIONS`, etc).

Aggregations

Django has out-of-the-box support for aggregation. The following aggregations are currently supported by Django MongoDB Engine:

- `Count`
- `Avg`
- `Min`
- `Max`
- `Sum`

MongoDB's group command is used to perform aggregations using generated Javascript code that implements the aggregation functions.

While being more flexible than *Map/Reduce*, a `group` command can not be processed in parallel, for which reason you should prefer Map/Reduce to process big data sets.

Warning: Needless to say, you shouldn't use these aggregations on a regular basis (i.e. in your views or business logic) but regard them as a powerful tool for one-time operations.

Lower-Level Operations

When you hit the limit of what's possible with Django's ORM, you can always go down one abstraction layer to PyMongo.

You can use *raw queries and updates* to update or query for model instances using raw Mongo queries, bypassing Django's model query APIs.

If that isn't enough, you can skip the model layer entirely and operate on *PyMongo-level objects*.

Warning: These APIs are available for MongoDB only, so using any of these features breaks portability to other non-relational databases (Google App Engine, Cassandra, Redis, ...). For the sake of portability you should try to avoid database-specific features whenever possible.

Raw Queries and Updates

`MongoDBManager` provides two methods, `raw_query()` and `raw_update()`, that let you perform raw Mongo queries.

Note: When writing raw queries, please keep in mind that no field name substitution will be done, meaning that you'll always have to use database-level names – e.g. *_id* instead of *id* or *foo_id* instead of *foo* for foreignkeys.

Raw Queries

`raw_query()` takes one argument, the Mongo query to execute, and returns a standard Django queryset – which means that it also supports indexing and further manipulation.

As an example, let's do some Geo querying.

```
from djangotoolbox.fields import EmbeddedModelField
from django_mongodb_engine.contrib import MongoDBManager

class Point(models.Model):
    latitude = models.FloatField()
    longtitude = models.FloatField()

class Place(models.Model):
    ...
    location = EmbeddedModelField(Point)

    objects = MongoDBManager()
```

To find all places near to your current location, 42°N | π°E, you can use this raw query:

```
>>> here = {'latitude' : 42, 'longtitude' : 3.14}
>>> Place.objects.raw_query({'location' : {'$near' : here}})
```

As stated above, `raw_query()` returns a standard Django queryset, for which reason you can have even more fun with raw queries:

```
Limit the number of results to 10
>>> Foo.objects.raw_query({'location' : ...})[:10]

Keep track of most interesting places
>>> Foo.objects.raw_query({'location' : ...}) \
...             .update(interest=F('interest')+1)

and whatnot.
```

Raw Updates

`raw_update()` comes into play when Django MongoDB Engine's atomic updates through `$set` and `$inc` (using F) are not powerful enough.

The first argument is the query which describes the subset of documents the update should be executed against - as Q object or Mongo query. The second argument is the update spec.

Consider this model:

```
from django_mongodb_engine.contrib import MongoDBManager

class FancyNumbers(models.Model):
    foo = models.IntegerField()

    objects = MongoDBManager()
```

Let's do some of those super-cool MongoDB in-place bitwise operations.

```
FancyNumbers.objects.raw_update({}, {'$bit' : {'foo' : {'or' : 42}}})
```

That bitwise-ORs every *foo* of all documents in the database with 42.

To run that update against a subset of the documents, for example against any whose *foo* is greater than π, use a non-empty filter condition:

```
FancyNumbers.objects.raw_update(Q(foo__gt=3.14), {'$bit' : ...})
# or
FancyNumbers.objects.raw_update({'foo' : {'$gt' : 3.14}}, {'$bit' : ...})
```

PyMongo-level

`django.db.connections` is a dictionary-like object that holds all database connections – that is, for MongoDB databases, `django_mongodb_engine.base.DatabaseWrapper` instances.

These instances can be used to get the PyMongo-level `Connection`, `Database` and `Collection` objects.

For example, to execute a `find_and_modify()` command, you could use code similar to this:

```
from django.db import connections
database_wrapper = connections['my_db_alias']
eggs_collection = database_wrapper.get_collection('eggs')
eggs_collection.find_and_modify(...)
```

Reference

Fields

This is a reference of both fields that are implemented in djangotoolbox and fields specific to MongoDB.

(In signatures, `...` represents arbitrary positional and keyword arguments that are passed to `django.db.models.Field`.)

in `djangotoolbox`

in `django_mongodb_engine`

GridFS Storage

Map/Reduce

```
from django_mongodb_engine.contrib import MongoDBManager

class MapReduceableModel(models.Model):
    ...
    objects = MongoDBManager()
```

```
>>> MapReduceableModel.objects.filter(...).map_reduce(...)
```

Settings

Client Settings

Additional flags may be passed to `pymongo.MongoClient` using the `OPTIONS` dictionary:

```
DATABASES = {
    'default' : {
        'ENGINE' : 'django_mongodb_engine',
        'NAME' : 'my_database',
        ...
        'OPTIONS' : {
            'socketTimeoutMS' : 500,
            ...
        }
    }
}
```

All of these settings directly mirror PyMongo settings. In fact, all Django MongoDB Engine does is lower-casing the names before passing the flags to `MongoClient`. For a list of possible options head over to the PyMongo documentation on client options.

Acknowledged Operations

Use the `OPERATIONS` dict to specify extra flags passed to `Collection.save`, `update()` or `remove()` (and thus included in the write concern):

```
'OPTIONS' : {
    'OPERATIONS' : {'w' : 3},
    ...
}
```

Get a more fine-grained setup by introducing another layer to this dict:

```
'OPTIONS' : {
    'OPERATIONS' : {
        'save' : {'w' : 3},
        'update' : {},
        'delete' : {'j' : True}
    },
    ...
}
```

Note: This operations map to the **Django** operations *save*, *update* and *delete* (**not** to MongoDB operations). This is because Django abstracts "*insert vs. update*" into *save*.

A full list of write concern flags may be found in the MongoDB documentation.

Model Options

In addition to Django's default Meta options, Django MongoDB Engine supports various options specific to MongoDB through a special `class MongoMeta`.

```
class FooModel(models.Model):
    ...
    class MongoMeta:
        # Mongo options here
        ...
```

Indexes

Django MongoDB Engine already understands the standard `db_index` and `unique_together` options and generates the corresponding MongoDB indexes on `syncdb`.

To make use of other index features, like multi-key indexes and Geospatial Indexing, additional indexes can be specified using the `indexes` setting.

```
class Club(models.Model):
   location = ListField()
   rating = models.FloatField()
   admission = models.IntegerField()
   ...
   class MongoMeta:
      indexes = [
         [('rating', -1)],
         [('rating', -1), ('admission', 1)],
         {'fields': [('location', '2d')], 'min': -42, 'max': 42},
      ]
```

`indexes` can be specified in two ways:

- The simple "without options" form is a list of `(field, direction)` pairs. For example, a single ascending index (the same thing you get using `db_index`) is expressed as `[(field, 1)]`. A multi-key, descending index can be written as `[(field1, -1), (field2, -1), ...]`.
- The second form is slightly more verbose but takes additional MongoDB index options. A descending, sparse index for instance may be expressed as `{'fields': [(field, -1)], 'sparse': True}`.

Capped Collections

Use the `capped` option and `collection_size` (and/or `collection_max`) to limit a collection in size (and/or document count), new documents replacing old ones after reaching one of the limit sets.

For example, a logging collection fixed to 50MiB could be defined as follows:

```
class LogEntry(models.Model):
    timestamp = models.DateTimeField()
    message = models.TextField()
    ...
    class MongoMeta:
        capped = True
        collection_size = 50*1024*1024
```

Warning: These APIs are available for MongoDB only, so using any of these features breaks portability to other non-relational databases (Google App Engine, Cassandra, Redis, ...). For the sake of portability you should try to avoid database-specific features whenever possible.

Lower-Level API

```
from django_mongodb_engine.contrib import MongoDBManager

class FooModel(models.Model):
```

```
    ...
    objects = MongoDBManager()
```

```
>>> FooModel.objects.raw_query(...)
>>> FooModel.objects.raw_update(...)
```

Troubleshooting

This page is going to be a collection of common issues Django MongoDB Engine users faced. Please help grow this collection – *tell us about your troubles*!

`SITE_ID` issues

```
AutoField (default primary key) values must be strings representing an ObjectId on␣
↪MongoDB (got u'1' instead). Please make sure your SITE_ID contains a valid ObjectId␣
↪string.
```

This means that your `SITE_ID` setting (What's SITE_ID?!) is incorrect – it is set to "1" but the site object that has automatically been created has an ObjectId primary key.

If you add `'django_mongodb_engine'` to your list of `INSTALLED_APPS`, you can use the `tellsiteid` command to get the default site's ObjectId and update your `SITE_ID` setting accordingly:

```
$ ./manage.py tellsiteid
The default site's ID is u'deafbeefdeadbeef00000000'. To use the sites framework, add␣
↪this line to settings.py:
SITE_ID=u'deafbeefdeadbeef00000000'
```

Creating/editing user in admin causes `DatabaseError`

```
DatabaseError at /admin/auth/user/deafbeefdeadbeef00000000/
[...] This query is not supported by the database.
```

This happens because Django tries to execute JOINs in order to display a list of groups/permissions in the user edit form.

To workaround this problem, add `'djangotoolbox'` to your `INSTALLED_APPS` which makes the Django admin skip the groups and permissions widgets.

No form field implemented for <class 'djangotoolbox.fields.ListField'>

See https://gist.github.com/1200165

Meta

Changelog

Version 0.6 (Jul 12, 2015)

- Add support for PyMongo 3 (Thanks @markunsworth & @ajdavis)

Version 0.5.2 (Jun 19, 2015)

- Add support for Replica Sets (Thanks @r4fek)
- Make safe writes the default (Thanks @markunsworth)

Version 0.5.1 (Nov 2013)

- Fixed packaging issues

Version 0.5 (Nov 2013)

Major changes

- Added support for Django 1.4-1.6, requires djangotoolbox >= 1.6.0
- PyPy support
- MongoDB 2.0 support
- We're now on Travis
- New custom primary key behavior (to be documented)
- New `MongoMeta.indexes` system (see *Model Options*), deprecation of `MongoMeta.{index_together,descending_indexes,sparse_indexes}`

Minor changes/fixes

- Support for MongoDB `distinct()` queries
- Support for reversed-`$natural` ordering using `reverse()`
- Dropped `LegacyEmbeddedModelField`
- `url()` support for the GridFS Storage
- Deprecation of `A()` queries
- Deprecation of the `GridFSField.versioning` feature
- Numerous query generator fixes
- Fixed `DecimalField` values sorting
- Other bug fixes, cleanup, new tests etc.

Version 0.4 (May 2011)

- *GridFS storage backend*
- Fulltext search
- Query logging support
- Support for sparse indexes (see *Model Options*)
- Database settings specific to MongoDB were moved into the `OPTIONS` dict. (see *Settings*) Furthermore, the *SAFE_INSERTS* and *WAIT_FOR_SLAVES* flags are now deprecated in favor of the new `OPERATIONS` setting (see *Acknowledged Operations*)
- Added the *tellsiteid command*
- Defined a stable *lower-level database API*
- Numerous bug fixes, new tests, code improvements and deprecations

Version 0.3 (Jan 2011)

- *OR* query support
- Support for `DateTimeField` and friends
- Support for atomic updates using F
- `EmbeddedModelField` has been merged into djangotoolbox. For legacy data records in your setup, you can use the `LegacyEmbeddedModelField`.
- Support for *raw queries and raw updates*

Version 0.2 (Oct 2010)

- *Aggregation support*
- *Map/Reduce support*
- `ListField`, `SetListField`, `DictField` and `GenericField` have been merged into djangotoolbox
- Added an `EmbeddedModelField` to store arbitrary model instances as MongoDB embedded objects/subobjects.
- Internal Refactorings

How to Contribute

We'd love to see you getting involved in Django MongoDB Engine's development!

Here are some ideas on how you can help evolve this project:

- *Send us feedback*! Tell us about your good or bad experiences with Django MongoDB Engine – what did you like and what should be improved?
- Blog/write about using Django with MongoDB and *let us know about your work*.
- Help solving other people's problems on *the mailing list*.
- Fix and improve this documentation. Since none of the Django MongoDB Engine developers are native speakers, these documents are probably full of typos and weird, ungrammatical phrasings. Also, if you're missing something from this documentation, please tell us!

- *Report bugs and feature requests*.
- *Send patches or pull requests* containing bug fixes, new features or code improvement.

Mailing List

Our mailing list, `django-non-relational@googlegroups.com`, is the right place for general feedback, discussion and support.

Development

Django MongoDB Engine is being developed on GitHub.

Bug Reports

Bugs can be reported to our ticket tracker on GitHub.

Patches

The most comfortable way to get your changes into Django MongoDB Engine is to use GitHub's pull requests. It's perfectly fine, however, to send regular patches to *the mailing list*.

Authors

Current Primary Authors

- Jonas Haag <jonas@lophus.org>
- Flavio Percoco Premoli <flaper87@flaper87.org>
- Wojtek Ruszczewski <github@wr.waw.pl>

Past Primary Authors

- Alberto Paro <alberto@ingparo.it>
- George Karpenkov <george@metaworld.ru>

Contributions by

- Ellie Frost (https://github.com/stillinbeta)
- Simon Charette (https://github.com/charettes)
- Dag Stockstad (https://github.com/dstockstad) – found tons of bugs in our query generator
- Shane R. Spencer (https://github.com/whardier) – Website Review
- Sabin Iacob (https://github.com/m0n5t3r)
- kryton (https://github.com/kryton)
- Brandon Pedersen (https://github.com/bpedman)

(For an up-to-date list of contributors, see https://github.com/django-nonrel/mongodb-engine/contributors.)