django-mongonaut Documentation

Release 0.2.20

Daniel Greenfeld

Aug 17, 2017

Contents

This is an introspective interface for Django and MongoDB. Built from scratch to replicate some of the Django admin functionality, but for MongoDB.

Contents:

CHAPTER 1

Installation

Normal Installation

Get MongoDB:

```
Download the right version per http://www.mongodb.org/downloads
```

Get the code:

```
pip install django-mongonaut==0.2.20
```

Install the dependency in your settings file (settings.py):

```
INSTALLED_APPS = (
    ...
    'mongonaut',
    ...
)
```

Add the mongonaut urls.py file to your urlconf file:

```
urlpatterns = patterns('',
    ...
    url(r'^mongonaut/', include('mongonaut.urls')),
    ...
)
```

Also in your settings file, you'll need something like:

```
# mongodb connection
from mongoengine import connect
connect('example_blog')
```

You will need the following also set up:

- django.contrib.sessions
- django.contrib.messages

Note: No need for *autodiscovery()* with django-mongonaut!

Static Media Installation

By default, *django-mongonaut* uses static media hosted by other services such as Google or Github. If you need to point to another location, then you can change the following defaults to your new source:

```
# settings.py defaults
MONGONAUT_JQUERY = "http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"
MONGONAUT_TWITTER_BOOTSTRAP = "http://twitter.github.com/bootstrap/assets/css/
↪bootstrap.css"
MONGONAUT_TWITTER_BOOTSTRAP_ALERT = http://twitter.github.com/bootstrap/assets/js/
↪bootstrap-alert.js"
```

Heroku MongoDB connection via MongoLabs

Your connection string will be provided by MongoLabs in the Heroku config. To make that work, just use the following code instead the *# mongodb connection* example:

```
# in your settings file (settings.py)
import os
import re
from mongoengine import connect
regex = re.compile(r'^mongodb\:\/\/(?P<username>[_\w]+):(?P<password>[\w]+)@(?P<host>
↪[\.\w]+):(?P<port>\d+)/(?P<database>[_\w]+)$')
mongolab_url = os.environ['MONGOLAB_URI']
match = regex.search(mongolab_url)
data = match.groupdict()
connect(data['database'], host=data['host'], port=int(data['port']), username=data[
↪'username'], password=data['password'])
```

CHAPTER 2

Configuration

One of the most useful parts of *django.contrib.admin* is the ability to configure various views that touch and alter data. django-mongonaut is similar to the Django Admin, but adds in new functionality and ignores other features. The reason is that MongoDB is not a relational database, so attempting to replicate in general simply removes some of the more useful features we get from NoSQL.

Basic Pattern

In your app, create a module called 'mongoadmin.py'. It has to be called that or django-mongonaut will not be able to find it. Then, in your new mongonaut file, simply import the mongoengine powered models you want mongonaut to touch, then import the MongoAdmin class, instantiate it, and finally attach it to your model.

```
# myapp/mongoadmin.py

# Import the MongoAdmin base class
from mongonaut.sites import MongoAdmin

# Import your custom models
from blog.models import Post

# Instantiate the MongoAdmin class
# Then attach the mongoadmin to your model
Post.mongoadmin = MongoAdmin()
```

That's it! Now you can view, add, edit, and delete your MongoDB models!

Note: You will notice a difference between how and *django.contrib.admin* and *django-mongonaut* do configuration. The former associates the configuration class with the model object via a registration utility, and the latter does so by adding the configuration class as an attribute of the model object.

More details and features are available in the API reference document.

CHAPTER 3

API

The following are advanced configuration features for django-mongonaut. Using them requires you to subclass the mongonaut.MongoAdmin class, then instantiate and attach your subclass as an attribute to the MongoEngine model.

Note: Future versions of mongonaut will allow you to work with pymongo collections without mongoengine serving as an intermediary.

MongoAdmin Objects

class MongoAdmin

The MongoAdmin class is the representation of a model in the mongonaut interface. These are stored in a file named mongoadmin.py in your application. Let's take a look at a very simple example of the MongoAdmin:

```
# myapp/mongoadmin.py

# Import the MongoAdmin base class
from mongonaut.sites import MongoAdmin

# Import your custom models
from blog.models import Post

# Subclass MongoAdmin and add a customization
class PostAdmin(MongoAdmin):

    # Searches on the title field. Displayed in the DocumentListView.
    search_fields = ('title',)

    # provide following fields for view in the DocumentListView
    list_fields = ('title', "published", "pub_date")
```

```
# Instantiate the PostAdmin subclass
# Then attach PostAdmin to your model
Post.mongoadmin = PostAdmin()
```

MongoAdmin Options

The MongoAdmin is very flexible. It has many options for dealing with customizing the interface. All options are defined on the MongoAdmin subclass:

has_add_permission

default:

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    def has_add_permission(self, request):
        """ Can add this object """
        return request.user.is_authenticated and request.user.is_active and request.
↪user.is_staff)
```

has_edit_permission

default:

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    def has_delete_permission(self, request):
        """ Can delete this object """
        return request.user.is_authenticated and request.user.is_active and request.
↪user.is_admin()
```

has_edit_permission

default:

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    def has_edit_permission(self, request):
        """ Can edit this object """
        return request.user.is_authenticated and request.user.is_active and request.
↪user.is_staff)
```

has_view_permission

default:

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    def has_view_permission(self, request):
        """ Can view this object """
        return request.user.is_authenticated and request.user.is_active
```

list_fields

default: Mongo _id

Accepts an iterable of string fields that matches fields in the associated model. Displays these fields as columns.

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    # provide following fields for view in the DocumentListView
    list_fields = ('title', "published", "pub_date")
```

search_fields

default: []

Accepts an iterable of string fields that matches fields in the associated model. Displays a search field in the DocumentListView. Performs an 'icontains' search with an 'OR' between evaluations.

```
# myapp/mongoadmin.py
class PostAdmin(MongoAdmin):

    # Searches on the title field. Displayed in the DocumentListView.
    search_fields = ('title',)
```

CHAPTER 4

Future Usage Concepts

Warning: This is is not implemented. It's sort of my dream of what I want this project to be able to do.

Complex version. Create a mongonaut.py module in your app:

```
#myapp.mongonaut
from datetime import datetime

from mongonaut.sites import MongoAdmin

from blog.models import Post

class ArticleAdmin(MongoAdmin):

    search_fields = ['title',]

    #This shows up on the DocumentListView of the Posts
    list_actions = [publish_all_drafts,]

    # This shows up in the DocumentDetailView of the Posts.
    document_actions = [generate_word_count,]

    field_actions = {confirm_images: 'image'}

    def publish_all_drafts(self):
        """ This shows up on the DocumentListView of the Posts """
        for post in Post.objects.filter(published=False):
            post.published = True
            post.pub_date = datetime.now()
            post.save()

    def generate_word_count(self):
        """ This shows up in the DocumentDetailView of the Posts.
        ID in this case is somehow the ID of the Posting objecy
```

```
        """
        return len(Post.objects.get(self.id).content.split(' '))

    def confirm_images(self):
        """ This will be attached to a field in the generated form
            specified in a dictionary
        """
        do_xyz()
        # TODO write this code or something like it

Article.mongoadmin = ArticleAdmin()
```

. include:: ../contributing.md

CHAPTER 5

Features

Introspection of mongoengine data

- Introspection via mongo engine
- Q based searches
- django.contrib.admin style browsing
- Automatic detection of field types
- Automatic discovery of collections

Introspection of pymongodata

- **[in progress]** Admin determination of which fields are displayed. Currently they can do so in the Document List view but not the Document Detail view.
- **[in progress]** Introspection via pymongo. This is becoming very necessary. Plan:
  - Always guarantee the _id.
  - Allow devs to set 1 or more field as 'expected'. But there is no hard contract!
  - introspect on field types to match how pymongo pulls data. So a *str* is handled differently than a list field.

Data Management

- **[in progress]** Admin authored Collection level document control functions
- EmbeddedDocumentsFields
- Editing on ListFields

- Document Deletes
- Editing on most other fields including ReferenceFields.
- Automatic detection of widget types
- Text field shorthand for letting user quickly determine type when using without mongoengine
- Document Adds

Permissions

- **[in progress]** Group defined controls
- User level controls
- Staff level controls
- Admin defined controls

CHAPTER 6

Indices and tables

- genindex
- modindex
- search