Django Dockit Documentation

Release 0.0.13b0

Jason Kraus

August 27, 2015

Contents

Django-DocKit provides a Document ORM in django. DocKit attempts to provide a batteries included experience while preserving django's various conventions.

CHAPTER 1

Installation

1.1 Requirements

- Python 2.6 or later
- Django 1.3

1.2 Settings

Put 'dockit' into your `INSTALLED_APPS` section of your settings file.

1.2.1 Configuring Document Store Backend

Django Document

Set the following in your settings file:

```
DOCKIT_BACKENDS = {
    'default': {
        'ENGINE': 'dockit.backends.djangodocument.backend.ModelDocumentStorage',
    }
}
DOCKIT_INDEX_BACKENDS = {
    'default': {
        'ENGINE': 'dockit.backends.djangodocument.backend.ModelIndexStorage',
    },
}

#Uncomment to use django-ztask for indexing
#DOCKIT_INDEX_BACKENDS['default']['INDEX_TASKS'] = 'dockit.backends.djangodocument.tasks.ZTaskIndexTa

#Uncomment to use django-celery for indexing
#DOCKIT_INDEX_BACKENDS['default']['INDEX_TASKS'] = 'dockit.backends.djangodocument.tasks.CeleryIndexT
```

Then add 'dockit.backends.djangodocument' to `INSTALLED_APPS`

Mongodb

Set the following in your settings file:

```
DOCKIT_BACKENDS = {
    'default': {
        'ENGINE':'dockit.backends.mongo.backend.MongoDocumentStorage',
        'USER':'travis',
        'PASSWORD':'test',
        'DB':'mydb_test',
        'HOST':'127.0.0.1',
        'PORT':27017,
    }
}
DOCKIT_INDEX_BACKENDS = {
    'default': {
        'ENGINE':'dockit.backends.mongo.backend.MongoIndexStorage',
        'USER':'travis',
        'PASSWORD':'test',
        'DB':'mydb_test',
        'HOST':'127.0.0.1',
        'PORT':27017,
    },
}
```

CHAPTER 2

Hyperadmin

The primary way to use dockit with an Admin interface and APIs is to use Django-Hyperadmin. Install django-hyperadmin-dockitresource (https://github.com/zbyte64/django-hyperadmin-dockitresource) to start using DocKit with Hyperadmin.

CHAPTER 3

Creating Indexes

Creating indexes in dockit is allot like constructing a django query except that no joins are currently allowed. After constructing a query, calling commit() ensures that the database has an index covering the criteria you supplied.

Examples:

```
#to create the index
MyDocument.objects.filter(published=True).commit()
#to use the index
for doc in MyDocument.objects.filter(published=True):
    print doc

#create an index for published documents that index the slug field
MyDocument.objects.filter(published=True).index('slug').commit()
#to use the index
MyDocument.objects.filter(published=True).get(slug='this-slug')

#index with a date
MyDocument.objects.filter(published=True).index('publish_date')

MyDocument.objects.filter(published=True).filter(publish_date__lte=datetime.datetime.now())
```

CHAPTER 4

Views

Dockit provides all the generic class based views that django provides but re-tooled for documents.

TODO

CHAPTER 5

Validation

Documents and Schemas support validation that mimics Django's models. Given a document instance you may call *full_clean* to validate the document structure and have a *ValidationError* raised if the document does not conform. Documents and schemas may define their own *clean_<fieldname>* method to validate each entry and a *clean* method to validate the entire document. All validation errors are to be sublcassed from *django.core.exceptions.ValidationError*.

To run through the document validation run *full_clean*:

```
try:
    mydocument.full_clean()
except ValidationError as e:
    print e
    raise
```

Adding custom validation to a document:

```
class MyDocument(schema.Document):
    full_name = schema.CharField()

    def clean_full_name(self):
        value = self.full_name.strip()
        if ' ' not full_name:
            raise ValidationError('A full name must have a first and last name')
        return full_name.strip()

    def clean(self):
        if datetime.date.today().weekday() == 2:
            raise ValidationError('You cannot validate on a Wednesday!')
```

CHAPTER 6

Multiple Record Types

Documents and Schemas maybe subclassed to provide polymorphic functionality. For this to work the base class must define *typed_field* in its Meta class which specifies a field name to store the type of schema. The subclasses must define *typed_key* which is to be a unique string value identifying that sublcass.

Here is an example where a collection contains multiple record types:

```
class ParentDocument(schema.Document):
    slug = schema.SlugField()

    class Meta:
        typed_field = '_doctype'

class Blog(ParentDocument):
    author = schema.CharField()
    body = schema.TextField()

    class Meta:
        typed_key = 'blog'

class Video(ParentDocument):
    url = schema.CharField()
    thumbnail = schema.ImageField(blank=True, null=True)

    class Meta:
        typed_key = 'video'

Blog(slug='blog-entry', author='John Smith', body='large description').save()
Video(slug='a-video', url='http://videos/url').save()

ParentDocument.objects.all() #an iterator containing a Blog and Video entry
```

Embedded schemas may also take advantage of this functionality as well:

```
class Download(schema.Schema):
    class Meta:
        typed_field = '_dltype'

class Bucket(schema.Document):
    slug = schema.SlugField()
    downloads = schema.ListField(schema.SchemaField(Download))

class Image(Download):
    full_image = schema.ImageField()
```

```
    class Meta:
        typed_key = 'image'

class Video(Download):
    url = schema.CharField()
    thumbnail = schema.ImageField(blank=True, null=True)

    class Meta:
        typed_key = 'video'

bucket = Bucket(slug='my-bucket')
bucket.downloads.append(Image(full_image=myfile))
bucket.downloads.append(Video(url='http://videos/url'))
bucket.save()
```

When we retrieve our bucket later we can expect the entries in downloads to be appropriately mapped to Image and Video. Outside applications may also add other Download record types and our Bucket class will be made aware of those types.

CHAPTER 7

Fixtures

Fixture handling in Django-DocKit is handled by Django-DataTap [https://github.com/zbyte64/django-datatap]. With this library DocKit provides fixture loading and dumping from the command line.

usage:

```
manage.py datatap Document [<app label>, ...] [<collection name>, ...]
```

CHAPTER 8

Internal API Reference

8.1 Backends

8.1.1 Base Backend

class dockit.backends.base.**BaseDocumentQuerySet**

The *BaseDocumentQuerySet* class provides an interface for implementing document querysets.

__len__()
returns an integer representing the number of items in the queryset

count()
Alias of __len__

delete()
deletes all the documents in the given queryset

all()
Returns a copy of this queryset, minus any caching.

get(*doc_id*)
returns the documents with the given id belonging to the queryset

__getitem__(*val*)
returns a document or a slice of documents

__nonzero__()
returns True if the queryset is not empty

class dockit.backends.base.**BaseDocumentStorage**

The *BaseDocumentStorage* class provides an interface for implementing document storage backends.

get_query(*query_index*)
return an implemented *BaseDocumentQuerySet* that contains all the documents

register_document(*document*)
is called for every document registered in the system

save(*doc_class*, *collection*, *data*)
stores the given primitive data in the specified collection

get(*doc_class*, *collection*, *doc_id*)
returns the primitive data for the document belonging in the specified collection

delete(*doc_class*, *collection*, *doc_id*)
deletes the given document from the specified collection

get_id(*data*)
returns the id from the primitive data

get_id_field_name()
returns a string representing the primary key field name

class dockit.backends.base.**BaseIndexStorage**
The *BaseIndexStorage* class provides an interface for implementing index storage backends.

register_index(*query_index*)
register the query index with this backend

destroy_index(*query_index*)
release the query index with this backend

get_query(*query_index*)
returns an implemented *BaseDocumentQuerySet* representing the query index

register_document(*document*)
is called for every document registered in the system

on_save(*doc_class*, *collection*, *doc_id*, *data*)
is called for every document save

on_delete(*doc_class*, *collection*, *doc_id*)
is called for every document delete

8.1.2 Builtin Backends

Mongo Backend

TODO

Django Document Backend

8.2 Queryset

class dockit.backends.queryset.**BaseDocumentQuery**(*query_index*, *backend=None*)
Implemented by the backend to execute a certain index

class dockit.backends.queryset.**QuerySet**(*query*)
Acts as a caching layer around an implemented *BaseDocumentQuery* TODO: implement actual caching

8.3 Schemas

8.3.1 The Schema Class

8.3.2 The Document Class

8.3.3 The Document Manager

8.4 Schema Field Reference

8.4.1 BaseField

8.4.2 BaseTypedField

8.4.3 CharField

8.4.4 TextField

8.4.5 IntegerField

8.4.6 BigIntegerField

8.4.7 BooleanField

8.4.8 DateField

8.4.9 DateTimeField

8.4.10 DecimalField

8.4.11 EmailField

8.4.12 FloatField

8.4.13 IPAddressField

8.4.14 PositiveIntegerField

8.4.15 SlugField

8.4.16 TimeField

8.4.17 SchemaTypeField

8.4.18 SchemaField

8.4.19 GenericSchemaField

8.4.20 TypedSchemaField

8.4.21 ListField

8.4.22 SetField

8.5.1 DocumentForm

TODO

Meta Options

- document
- schema
- dotpath
- exclude

TODO document the other options

8.6 Indexers

8.6.1 The Base Indexer Class

class `BaseIndexer`

The `BaseIndexer` class provides is an abstract class whose implementation provides querying functionality. An indexer is instantiated with the document, name and params.

8.7 Datataps

8.7.1 DocumentDataTap

CHAPTER 9

Contributing

DocKit is open-source and is here to serve the community by the community.

9.1 Philosophy

- **DocKit is BSD-licensed. All contributed code must be either**
  - the original work of the author contributed under the BSD license
  - work taken form another BSD-compatible license
- GPL or proprietary works are not eligible for contribution
- Master branch represents the current stable release
- Develop branch represents patches accepted for the next release

9.2 Reporting Issues

- Issues are tracked on github.
- Please ensure that a ticket hasn't already been submitted before submitting a ticket.
- **Tickets should include:**
  - A description of the problem
  - How to recreate the bug
  - Version numbers of the relevant packages in your Django stack
  - A pull request with a unit test exemplifying the failure is a plus

9.3 Contributing Code

1. Fork it on github
2. Branch from the release and push to your github
3. Apply the changes you need
4. Add yourself to AUTHORS.rst

5. Once done, send a pull request (to the develop branch)

Your pull request / patch should be:

- clear
- works across all supported versions
- follows the existing code style (mostly PEP-8)
- comments included where needed
- test case if it is to fix a bug
- if it changes functionality then include documentation

CHAPTER 10

Admin

Django admin integration is no longer actively supported, please use django-hyperadmin if possible. It is the belief of the maintainers that the Django Admin poorly supports documents and that an alternative is better in the long run. If you must use the Django Admin with dockit and come accross an issue then patches will be accepted.

Dockit provides some basic admin functionality for documents. Additionally the admin allows for editing of nested schemas in your document and also allows for customization on a per schema basis.

CHAPTER 11

Release Notes

11.1 0.0.13

- Django Nose for testing [d639af9]
- Django 1.5 support [b671f83]
- Integrated with django-datataps, replaces manifests
- Improved natural key generation
- Query hashes now based on md5 [24a32f8]
- Added south migrations for djangodocument [6c88dc7]
- Fix for a reference field referencing itself [4a6ad36]

Download: http://github.com/webcube/django-dockit

CHAPTER 12

Help & Feedback

We have a mailing list for general discussion and help: http://groups.google.com/group/django-dockit/

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

d

Symbols

A

B

C

D

G

O

Q

R

S