
django-hstore Documentation

Release 1.4.2

nemeisdesign

March 02, 2017

1	Introduction	1
1.1	Features	1
1.2	Django Admin widget	1
1.3	Grappelli Admin widget	2
1.4	Limitations	2
2	Project Maturity	5
3	Deprecation policy	7
4	User Guide	9
4.1	Install	9
4.2	Setup	10
4.3	Usage	11
4.4	Python API	14
4.5	HSTORE manager	16
4.6	ReferenceField Usage	16
5	Developers Guide	19
5.1	Running tests	19
5.2	How to contribute	19
6	License	21

Introduction

django-hstore is a niche library which integrates the `hstore` extension of PostgreSQL into Django.

HStore brings the power of NoSQL key/value stores into PostgreSQL, giving us the advantage of flexibility and performance without renouncing to the robustness of SQL databases.

Mailing List: <https://groups.google.com/forum/#!forum/django-hstore>

Features

- Postgis compatibility.
- Python3 support.
- Nice admin widgets.
- Possibility to define a schema and use the standard django fields (since `django_hstore 1.3.0` and `django 1.6`)

Django Admin widget

django-hstore ships a nice admin widget that makes the field more user-friendly.

Name:	<input type="text" value="alpha"/>
Data:	
<input type="text" value="field1"/>	: <input type="text" value="value1"/> ✖
<input type="text" value="field2"/>	: <input type="text" value="value2"/> ✖
<input type="text" value="zip"/>	: <input type="text" value="00040"/> ✖
<input type="text" value="key"/>	: <input type="text" value="value"/> ✖
+ Add row ✎ toggle textarea	
✖ Delete Save and add another Save and continue editing Save	


Each time a key or a value is modified, the underlying textarea is updated:

Name:

Data:

Raw
textarea:

```
{
  "field1": "value1",
  "field2": "value2",
  "zip": "00040"
}
```

 toggle textarea

Grappelli Admin widget

If you use the awesome [django-grappelli](#) there's an even nicer looking widget for you too!

Extra data toggle textarea +

store extra attributes in JSON string

<input type="text" value="postal_code"/>	<input type="text" value="00030"/>	—
<input type="text" value="distance"/>	<input type="text" value="10"/>	—
<input type="text" value="name"/>	<input type="text" value="Federico"/>	—
<input type="text" value="key"/>	<input type="text" value="value"/>	—

Each time a key or a value is modified, the underlying textarea is updated:

Extra data toggle textarea +

store extra attributes in JSON string

Raw textarea

```
{
  "postal_code": "00030",
  "distance": "10",
  "name": "Federico"
}
```

Note: When using `SerializedDictionaryField`, data values are displayed in their serialized JSON form. This is done to make their type explicit.

Limitations

- PostgreSQL's implementation of hstore has no concept of type; it stores a mapping of string keys to string values. Values are stored as strings in the database regarding of their original type. **This limitation can be**

overcome by using either the schema mode since version 1.3.0 or by using the serialized dictionary field since version 1.3.6 of `django_hstore`.

- The hstore extension is not automatically installed on use with this package: you must install it manually.
- To run tests, hstore extension must be installed on `template1` database. To install hstore on `template1`:

```
$ psql -d template1 -c 'create extension hstore;'
```
- The admin widget will work with inlines only if using `StackedInline`. It won't work on `TabularInline`.
- If `django.middleware.transaction.TransactionMiddleware` is enabled and the project is deployed through `uwsgi`, the first request to a view working with models featuring hstore fields will raise an exception; see [Django Ticket #22297](#) for more details on this issue. This issue is specific to Django 1.6 and below.

Warning: Due to hstore being a postgresql extension and not a native type, its oid is different on each database, which causes strange behavior with type conversions (see more on [this issue](#)) if hstore extension is installed individually in each database.

To avoid this strange behavior you have two options:

- Install hstore on `template1` postgresql template database and recreate all databases/templates from it, which allows all database to have the same oid for the hstore type (this is the recommended way).
- Disable global registering setting `DJANGO_HSTORE_ADAPTER_REGISTRATION` by setting it to `connection` in your settings. This can have a performance impact because it registers the hstore extension for each new connection created (if you are using django 1.6, persistent connections - or any other connection pool - will help to reduce this impact).

Project Maturity

django-hstore is stable, widely used library with well defined deprecation policy.

Deprecation policy

At any moment of time, **django-hstore** developers will maintain support for three versions of django.

As example: The current stable release of django is 1.9, so **django-hstore** supports the following django versions: 1.9, 1.8 and 1.7. When django 1.10 is released, support for 1.7 will be dropped.

This section covers all aspects that user want know about **django-hstore**.

Install

This section covers a installing **django-hstore** and its requirements.

Requirements

- Python 2.7 or 3.3+
- Django 1.7, 1.8, 1.9
- Psycopg2 2.4.3+
- PostgreSQL 9.0+

Stable version

```
pip install django-hstore
```

Development version

```
pip install -e git+git://github.com/djangonauts/django-hstore#egg=django-hstore
```

Upgrade from older versions

In *version 1.2.x* some internals have been changed in order to simplify usage and prevent errors.

Values are automatically converted to strings, fields constantly validate input and so on.

If you are upgrading from an older version, ensure your application code works as expected. If it doesn't you will either have to update your code or tie your application's requirement to the older version of **django-hstore** (1.1.1).

Setup

Basic setup

First, add `django_hstore` to your `settings.INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    "django_hstore",
    ...
)
```

Second, collect static files (needed for the admin widget) with:

```
python manage.py collectstatic
```

Multiple database setup

If for some reason you have to use **django-hstore** in a *multi-database setup* and some of the database you are using don't have the hstore extension installed, you can skip hstore registration by setting `HAS_HSTORE` to `False` in your database config:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycpg2',
        'NAME': 'name',
        'USER': 'user',
        'PASSWORD': 'pass',
        'HOST': 'localhost',
        'PORT': '',
    },
    'other': {
        'ENGINE': 'django.db.backends.postgresql_psycpg2',
        'NAME': 'other',
        'USER': 'user',
        'PASSWORD': 'pass',
        'HOST': 'localhost',
        'PORT': '',
        'HAS_HSTORE': False,
    }
}
```

If you do that, then don't try to create `DictionaryField` in that database.

Be sure to check out [allow_syncdb](#) documentation.

Available Settings

- `DJANGO_HSTORE_ADAPTER_REGISTRATION`: defaults to `global`; set this to `connection` if you need compatibility with `SQLAlchemy`
- `DJANGO_HSTORE_ADAPTER_SIGNAL_WEAKREF`: the value of `weak` argument passed to the `connection_created` signal

Note to South users

If you keep getting errors like “There is no South database module ‘south.db.None’” for your database., add the following to settings.py:

```
SOUTH_DATABASE_ADAPTERS = {'default': 'south.db.postgresql_psycopg2'}
```

Usage

The library provides five principal classes:

- `django_hstore.hstore.DictionaryField` + An ORM field which stores a mapping of string key/value pairs in a hstore column.
- `django_hstore.hstore.SerializedDictionaryField` + Similar to the `DictionaryField` with the exception that all submitted values in string key/value are encoded-to JSON upon writes to the database and decoded from JSON upon database reads. This allows for any JSON supported data type to be stored in an hstore column.
- `django_hstore.hstore.ReferencesField` + An ORM field which builds on `DictionaryField` to store a mapping of string keys to django object references, much like `ForeignKey`.
- `django_hstore.hstore.HStoreManager` + An ORM manager which provides much of the query functionality of the library.
- `django_hstore.hstore.HStoreGeoManager` + An additional ORM manager to provide Geodjango functionality as well.

Model setup

the `DictionaryField` definition is straightforward:

```
from django.db import models
from django_hstore import hstore

class Something(models.Model):
    name = models.CharField(max_length=32)
    data = hstore.DictionaryField() # can pass attributes like null, blank, etc.

    objects = hstore.HStoreManager()
    # IF YOU ARE USING POSTGIS:
    # objects = hstore.HStoreGeoManager()
```

Since **django_hstore 1.3.0** it is possible to use the `DictionaryField` in **schema mode** in order to overcome the limit of values being only strings. Another advantage of using the schema mode is that you can recycle the standard django fields in the admin and hopefully elsewhere. **This feature is available only from django 1.6 onwards.**

To use the schema mode you just need to supply a schema parameter to the `DictionaryField`:

```
# models.py
from django.db import models
from django_hstore import hstore

class SomethingWithSchema(models.Model):
    name = models.CharField(max_length=32)
    data = hstore.DictionaryField(schema=[
        {
```

```

        'name': 'number',
        'class': 'IntegerField',
        'kwargs': {
            'default': 0
        }
    },
    {
        'name': 'float',
        'class': 'FloatField',
        'kwargs': {
            'default': 1.0
        }
    },
    {
        'name': 'char',
        'class': 'CharField',
        'kwargs': {
            'default': 'test', 'blank': True, 'max_length': 10
        }
    },
    {
        'name': 'text',
        'class': 'TextField',
        'kwargs': {
            'blank': True
        }
    },
    {
        'name': 'choice',
        'class': 'CharField',
        'kwargs': {
            'blank': True,
            'max_length': 10,
            'choices': (('choice1', 'choice1'), ('choice2', 'choice2'))
        }
    }
])

objects = hstore.HStoreManager()

```

After this declaration some additional virtual fields will be available in the model. Each virtual field will map to a key in the dictionary field, types are maintained behind the scenes by using the `to_python` method of the field class that has been specified for each key.

The `schema` parameter is a list of dictionaries, each dictionary representing a field.

Each dictionary should have the following keys:

name: indicates the name of the attribute that will be created on the model

class: the field class that will be used to create the virtual field, you can pass it a string and it will look into `django.db.models`, alternatively you can pass it a concrete class derived from `django.db.models.Field` imported from anywhere

kwargs: the keyword arguments that will be passed to the Field class. Common arguments are `verbose_name`, `max_length`, `blank`, `choices`, `default`.

The following standard django fields have been tested successfully:

- `IntegerField`

- FloatField
- DecimalField
- BooleanField
- CharField
- TextField
- DateField
- DateTimeField
- EmailField
- GenericIPAddressField
- URLField

Other fields might work as well except for FileField, ImageField, and BinaryField which would need some additional work.

The schema of a DictionaryField can be changed at run-time if needed by using the `reload_schema` method (introduced in version 1.3.4):

```
field = SchemaDataBag._meta.get_field('data')
# load a different schema
field.reload_schema([
    {
        'name': 'url',
        'class': 'URLField'
    }
])
# turn off schema mode
field.reload_schema(None)
```

the ReferenceField definition is also straightforward:

```
class ReferenceContainer(models.Model):
    name = models.CharField(max_length=32)
    refs = hstore.ReferencesField()

    objects = hstore.HStoreManager()
```

the SerializedDictionaryField definition is very similar to the standard dictionary field:

```
from django.db import models
from django_hstore import hstore

class Something(models.Model):
    name = models.CharField(max_length=32)
    data = hstore.SerializedDictionaryField() # can pass attributes like null, blank, etc.

    objects = hstore.HStoreManager()
    # IF YOU ARE USING POSTGIS:
    # objects = hstore.HStoreGeoManager()
```

Optionally, the data accepts both a serializer and deserializer argument (which default to `json.dumps` and `json.loads`, respectively). This allows allowing for customized manners of serialization. **Customizing the serializer/deserializer is only partially implemented. It is NOT supported with the default Django admin widget (which attempts to serialize and deserialize all values with “`json.dumps`” and “`json.loads`”). Use at your own risk.**

Python API

You then treat the `data` field as simply a dictionary of string pairs:

```
instance = Something.objects.create(name='something', data={'a': '1', 'b': '2'})
assert instance.data['a'] == '1'

empty = Something.objects.create(name='empty')
assert empty.data == {}

empty.data['a'] = '3'
empty.save()
assert Something.objects.get(name='empty').data['a'] == '3'
```

In **default mode**, Booleans, integers, floats, lists, and dictionaries will be converted to strings, while lists, dictionaries, and booleans are converted into JSON formatted strings, so can be decoded if needed:

```
instance = Something.objects.create(name='something', data={'int': 1, 'bool': True})

instance.data['int'] == '1'
instance.data['bool'] == 'true'

import json
instance.data['dict'] = { 'list': ['a', False, 1] }
instance.data['dict'] == '{"list": ["a", false, 1]}'
json.loads(instance.data['dict']) == { 'list': ['a', False, 1] }
# => True
```

Since version **1.3.0** you can use the **schema mode** and you will be able to use virtual fields derived from standard django fields which will take care of validation, default values, type casting, choices and so on. Each virtual field will be mapped to a key of the `DictionaryField`:

```
>>> obj = SomethingWithSchema()
>>> obj.number
0
>>> obj.float
1.0
>>> obj.number = 3
>>> obj.float = 9.99
>>> obj.save()
>>> obj = SomethingWithSchema.objects.last()
>>> obj.number
3
>>> obj.data['number']
3
>>> obj.float
9.99
>>> obj.data['float']
9.99
```

Since version **1.3.6** you can use the `SerializedDictionaryField` to store any data type support in JSON. This has the specific advantage over the schema mode of not requiring the user to specify schema ahead of time.

```
>>> obj = SerializedExample.objects.create(
...     name="A Serializable Field!",
...     data={
...         'str': 'A string',
...         'int': 1234,
...         'float': 3.141,
```

```

...     'bool': True,
...     'list': [0, 'one', [2.0, 2.1]],
...     'dict': {
...         'a': 1,
...         'b': 'two',
...         'c': ['three']
...     }
... }
... )

>>> obj.data
{'int': 1234, 'float': 3.141, 'list': [0, 'one', [2.0, 2.1]], 'bool': True, 'str': 'A string', 'dict':

```

You can issue indexed queries against hstore fields:

```

# equivalence
Something.objects.filter(data={'a': '1', 'b': '2'})

# comparison (greater than, less than or equal to, ecc)
Something.objects.filter(data__gt={'a': '1'})
Something.objects.filter(data__gte={'a': '1'})
Something.objects.filter(data__lt={'a': '2'})
Something.objects.filter(data__lte={'a': '2'})

# more than one key can be supplied, the result will include the objects which satisfy the
# condition (greater than, less than or equal to, ecc) on all supplied keys
Something.objects.filter(data__gt={'a': '1', 'b': '2'})
Something.objects.filter(data__gte={'a': '1', 'b': '2'})
Something.objects.filter(data__lt={'a': '2', 'b': '3'})
Something.objects.filter(data__lte={'a': '2', 'b': '3'})

# subset by key/value mapping
Something.objects.filter(data__contains={'a': '1'})

# subset by list of some key values
# Note: Incompatible with the SerializedDictionaryField (lists as values are treated as actual values)
Something.objects.filter(data__contains={'a': ['1', '2']})

# subset by list of keys
# Note: Incompatible with the SerializedDictionaryField (lists as values are treated as actual values)
Something.objects.filter(data__contains=['a', 'b'])

# subset by single key
# Note: Incompatible with the SerializedDictionaryField (lists as values are treated as actual values)
Something.objects.filter(data__contains=['a'])

# filter by is null on individual key/value pairs
Something.objects.filter(data__isnull={'a': True})
Something.objects.filter(data__isnull={'a': True, 'b': False})

# filter by is null on the column works as normal
Something.objects.filter(data__isnull=True)

```

You can still do classic django “contains” lookups as you would normally do for normal text fields if you were looking for a particular string. In this case, the HSTORE field will be converted to text and the lookup will be performed on all the keys and all the values:

```
Something.objects.create(data={ 'some_key': 'some crazy Value' })

# classic text lookup (look up for occurrence of string in all the keys)
Something.objects.filter(data__contains='crazy')
Something.objects.filter(data__contains='some_key')
# classic case insensitive text lookup
Something.objects.filter(data__icontains='value')
Something.objects.filter(data__icontains='SOME_KEY')
```

HSTORE manager

You can also take advantage of some db-side functionality by using the manager:

```
# identify the keys present in an hstore field
>>> Something.objects.hkeys(id=instance.id, attr='data')
['a', 'b']

# peek at a a named value within an hstore field
>>> Something.objects.hpeek(id=instance.id, attr='data', key='a')
'1'

# do the same, after filter
>>> Something.objects.filter(id=instance.id).hpeek(attr='data', key='a')
'1'

# remove a key/value pair from an hstore field
>>> Something.objects.filter(name='something').hremove('data', 'b')
```

The hstore methods on manager **pass** all keyword arguments aside **from** `attr` and `key` to `.filter()`.

ReferenceField Usage

ReferenceField is a field that allows to reference other database objects without using a classic ManyToMany relationship.

Here's an example with the `ReferenceContainer` model defined in the **Model fields** section:

```
r = ReferenceContainer(name='test')
r.refs['another_object'] = AnotherModel.objects.get(slug='another-object')
r.refs['some_object'] = AnotherModel.objects.get(slug='some-object')
r.save()

r = ReferenceContainer.objects.get(name='test')
r.refs['another_object']
'<AnotherModel: AnotherModel object>'
r.refs['some_object']
'<AnotherModel: AnotherModel some_object>'
```

The database is queried only when references are accessed directly. Once references have been retrieved they will be stored for any eventual subsequent access:

```
r = ReferenceContainer.objects.get(name='test')
# this won't query the database
```

```
r.refs
{ u'another_object': u'myapp.models.AnotherModel:1',
  u'some_object': u'myapp.models.AnotherModel:2' }

# this will query the database
r.refs['another_object']
'<AnotherModel: AnotherModel object>'

# retrieved reference is now visible also when calling the HStoreDict object:
r.refs
{ u'another_object': <AnotherModel: AnotherModel object>,
  u'some_object': u'myapp.models.AnotherModel:2' }
```

Developers Guide

This section covers everything that **django-hstore** developer / contributor want know.

Running tests

Assuming one has the dependencies installed, and a **PostgreSQL 9.0+** server up and running:

```
python runtests.py
```

By default the tests run with the postgis backend.

If you want to run the tests with psycopg2 backend you can do:

```
python runtests.py --settings=settings_psycopg
```

You might need to tweak the DB settings according to your DB configuration.

If you need to do so you can copy the file `local_settings.py.example` to `local_settings.py` and add your database tweaks on it. `local_settings.py` will be automatically imported in `settings.py`. The same applies for `local_settings_psycopg.py.example`, which will be imported in `local_settings_psycopg.py`.

If after running this command you get an **error** saying:

```
type "hstore" does not exist
```

Try this:

```
psql template1 -c 'create extension hstore;'
```

More details here on link: [PostgreSQL error type hstore does not exist](#).

How to contribute

1. Join the mailing List: [django-hstore mailing list](#) and announce your intentions.
2. Follow *Style Guide for Python Code* <<http://www.python.org/dev/peps/pep-0008/>>
3. Fork this repo
4. Write code
5. Write tests for your code

6. Ensure all tests pass
7. Ensure test coverage is not under 90%
8. Document your changes
9. Send pull request

License

Copyright (C) 2013-2014 Federico Capoano

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Original Author

=====

Copyright (C) 2011 Jordan McCoy

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.