Django Postgres Extensions Documentation

Release 0.9.2

Paul Martin

February 10, 2017

Contents:

CHAPTER 1

Feature Overview

1.1 Custom Postgres backend

The customized Postgres backend adds the following features:

- HStore Extension is automatically activated when a test database is created so you don't need to create a separate migration which is useful especially when building re-usable apps
- Uses a different update compiler which adds some functionality outlined in the ArrayField section below
- If db_index is set to True for an ArrayField, a GIN index will be created which is more useful than the default database index for arrays.
- Adds some extra operators to enable ANY and ALL lookups

1.2 ArrayField

The included ArrayField has been subclassed from django.contrib.postgres.fields.ArrayField to add extra features and is a drop-in replacement. To use this ArrayField. The customized Postgres ArrayField adds the following features:

- Get array values by index:
- Update array values by index:
- Added database functions for interacting with Arrays. These functions handle the provided arguments by automatically converting them to the required expressions.
- Add an array of values to an existing field. In this case the output_field is required to tell Django what db type to use for the array:
- Additional lookups have been added to the ArrayField to enable queries using the ANY and ALL database functions.
- Use either a split array field or a multiple choice field in a model form

1.3 HStoreField

The included HStoreField has been subclassed from django.contrib.postgres.fields.HStoreField to add extra features and is a drop-in replacement. To use this HStoreField:

The customized Postgres HStoreField adds the following features:

- Get hstore values by key
- Update hstore by specific keys, leaving any others untouched
- Added database functions for interacting with HStores, these functions handle the arguments by converting them to the correct expressions automatically.
- Nested form field for a better representation of hstore in a form, either by providing a list of keys or list of form fields.

1.4 JSONField

The included JSONField has been subclassed from django.contrib.postgres.fields.JSONField to add extra features and is a drop-in replacement. To use this JSONField:

The customized Postgres JSONField adds the following features:

- Get json values by key or key path
- Update JSON Field by specific keys, leaving any others untouched
- Delete JSONField by key or key path
- Extra database functions for interacting with JSONFields. These functions handle the arguments by converting them to the correct expressions automatically.
- The same NestedFormField and NestedFormWidget referred to above for HStore can also be used with a JSON Field by providing a list of fields.

1.5 ModelAdmin

For an example of how these fields can be configured in a modelform; take the following models.py:

```
from django.db import models
from django_postgres_extensions.models.fields import HStoreField, JSONField, ArrayField
from django_postgres_extensions.models.fields.related import ArrayManyToManyField
from django import forms
from django.contrib.postgres.forms import SplitArrayField
from django_postgres_extensions.forms.fields import NestedFormField

details_fields = (
    ('Brand', NestedFormField(keys=('Name', 'Country'))),
     ('Type', forms.CharField(max_length=25, required=False)),
     ('Colours', SplitArrayField(base_field=forms.CharField(max_length=10, required=False), size=10))
)

class Buyer(models.Model):
    time = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=20)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=15)
    keywords = ArrayField(models.CharField(max_length=20), default=[], form_size=10, blank=True)
    sports = ArrayField(models.CharField(max_length=20),default=[], blank=True, choices=(
    ('football', 'Football'), ('tennis', 'Tennis'), ('golf', 'Golf'), ('basketball', 'Basketball'),
    shipping = HStoreField(keys=('Address', 'City', 'Region', 'Country'), blank=True, default={})
    details = JSONField(fields=details_fields, blank=True, default={})
    buyers =  ArrayManyToManyField(Buyer)

    def __str__(self):
        return self.name

    @property
    def country(self):
        return self.shipping.get('Country', '')
```

And with admin.py:

```
from django.contrib import admin
from django_postgres_extensions.admin.options import PostgresAdmin
from models import Product, Buyer

class ProductAdmin(PostgresAdmin):
    filter_horizontal = ('buyers',)
    fields = ('name', 'keywords', 'sports', 'shipping', 'details', 'buyers')
    list_display = ('name', 'keywords', 'shipping', 'details', 'country')

admin.site.register(Buyer)
admin.site.register(Product, ProductAdmin)
```

The form field would look like this:

The list display would look like this:

1.6 Additional Queryset Methods

The app adds the format method to all querysets. This will defer a field and add an annotation with a different format. For example to return a hstorefield as json:

```
qs = Model.objects.all().format('description', HstoreToJSONBLoose)
```

1.7 Array Many To Many Field

The Array Many To Many Field is designed be a drop-in replacement of the normal Django Many To Many Field and thus replicates many of its features.

The Array Many To Many field supports the following features which replicate the API of the regular Many To Many Field:

- Descriptor queryset with add, remove, clear and set for both forward and reverse relationships
- Prefetch related for both forward and reverse relationships
- Lookups across relationships with filter for both forward and reverse relationships
- Lookups across relationships with exclude for forward relationships only

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Getting started

Warning: Although it generally should work on other versions, Django Postgres Extensions has been tested with Python 2.7.12, Python 3.6 and Django 1.10.5.

3.1 Installation

Install with `pip install django_postgres_extensions`

3.2 Setup project

In your settings.py, add 'django.contrib.postgres' and 'django_postgres_extensions' to the list of INSTALLED APPS and configure the database to use the included backend (subclassed from the default Django Postgres backend):

```
INSTALLED_APPS = [
    'django.contrib.contenttypes',
    'django.contrib.auth',
    'django.contrib.sites',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.admin.apps.SimpleAdminConfig',
    'django.contrib.staticfiles',
    'django.contrib.postgres',
    'django_postgres_extensions'
]

DATABASES = {
    'default': {
        'ENGINE': 'django_postgres_extensions.backends.postgresql',
        'NAME': 'db',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': '127.0.0.1',
        'PORT': 5432,
    }
}
```

CHAPTER 4

ArrayField

4.1 Basic Usage

To use the ArrayField:

```
from django.db import models
from django_postgres_extensions.models.fields import ArrayField
class Product(models.Model):
    tags = ArrayField(models.CharField(max_length=15), null=True, blank=True)
    moretags = ArrayField(models.CharField(max_length=15), null=True, blank=True)
```

4.2 Array Indexes

- Get array values by index:

```
from django_postgres_extensions.models.expressions import Index, SliceArray
obj = Product.objects.annotate(Index('tags', 1)).get()
print(obj.tags__1)
obj = Product.objects.annotate(tag_1=Index('tags', 1)).get()
print(obj.tag_1)
obj = Product.objects.annotate(SliceArray('tags', 0, 1)).get()
print(obj.tags__0_1)
```

- Update array values by index:

```
Product.objects.update(tags__2='Heavy Metal')
```

4.3 Database Functions

Various database functions are included for manipulating arrays:

- ArrayLength: returns the length of an array
- ArrayPosition: the position of an item in an array
- ArrayPositions: all positions of an item in an array
- ArrayAppend: Create an array value by adding a value to the end of an array field
- ArrayPrepend: Create an array value by adding a value to the start of an array field

- ArrayRemove: Create an array value by removing a value from an array field
- ArrayReplace: Create an array value by replacing one value with another in an array field
- ArrayCat: Combine the values of two separate ArrayFields

For more information on each of these functions, check the postgresql documentation. The provided arguments to each function are automatically converted to the required expressions:

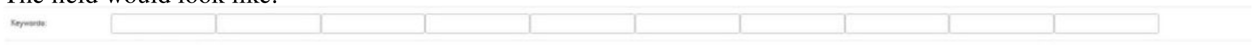
```
from django_postgres_extensions.models.functions import *
obj = Product.objects.queryset.annotate(tags_length=ArrayLength('tags', 1)).get()
obj = Product.objects.annotate(position=ArrayPosition('tags', 'Rock')).get()
obj = Product.objects.annotate(positions=ArrayPositions('tags', 'Rock')).get()
Product.objects.update(tags = ArrayAppend('tags', 'Popular'))
Product.objects.update(tags = ArrayPrepend('Popular', 'tags'))
Product.objects.update(tags = ArrayRemove('tags', 'Album'))
Product.objects.update(tags = ArrayReplace('tags', 'Rock', 'Heavy Metal'))
Product.objects.update(tags = ArrayCat('tags', 'moretags'))
Product.objects.update(tags=ArrayCat('tags', ['Popular', '8'], output_field=Product._meta.get_field(
```

4.4 Use in ModelForms

django.contrib.postgres includes two possible array form fields: SimpleArrayField (the default) and SplitArrayField. To use the SplitArrayField automatically when generating a ModelForm, add the form_size keyword argument to the ArrayField:

```
class Product(models.Model):
    keywords = ArrayField(models.CharField(max_length=20), default=[], form_size=10, blank=True)
```

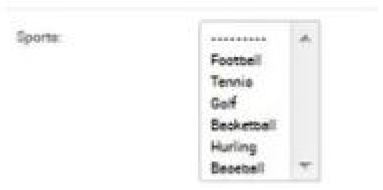
The field would look like:



Alternatively, it is possible to use a Multiple Choice Field for an Array, by specifying a choices argument:

```
sports = ArrayField(models.CharField(max_length=20),default=[], blank=True, choices=(
('football', 'Football'), ('tennis', 'Tennis'), ('golf', 'Golf'), ('basketball', 'Basketball'), ('hu
```

The field would look like:



4.5 Array Lookups

Additional lookups have been added to the ArrayField to enable queries using the ANY and ALL database functions:

```
qs = Product.objects.filter(tags__any = 'Popular')
qs = Product.objects.filter(tags_all__isstartof = 'Popular')
```

Any lookups check if any value in the array meets the lookup criteria. All lookups check is all values in an array meet the lookup criteria. The full list of additional lookups are:

- any
- any_exact
- any_gt
- any_gte
- any_lt
- any_lte
- any_in
- any_isstartof
- any_isendof
- any_contains (for 2d arrays)
- all
- all_exact
- all_gt
- all_gte
- all_lt
- all_lte
- all_in
- all_isstartof
- all_isendof
- all_regex

CHAPTER 5

HStoreField

5.1 Basic Usage

To use the HStoreField:

```
from django.db import models
from django_postgres_extensions.models.fields import HStoreField
class Product(models.Model):
    description = HStoreField(null=True, blank=True)
```

The customized Postgres HStoreField adds the following features:

5.2 Individual keys

- Get hstore values by key:

```
from django_postgres_extensions.models.expressions import Key, Keys
obj = Product.objects.annotate(Key('description', 'Release')).get()
obj = Product.objects.annotate(Keys('description', ['Industry', 'Release'])).get()
```

- Update hstore by specific keys, leaving any others untouched:

```
Product.objects.update(description__ = {'Genre': 'Heavy Metal', 'Popularity': 'Very Popular'})
```

5.3 Database functions

Various database functions are included for interacting with HStores.

- Slice: Return a dictionary with just the specified keys
- Delete: Delete a key or list of keys from the hstore. Keys can also be deleted by specifying a dictionary
- AKeys: Returns the hstore keys as a list
- AVals: Returns the hstore values as a list
- HStoreToArray: Returns the hstore as an array
- HStoreToMatrix: Returns the hstore as a matrix

- HstoreToJSONB: Returns the hstore as JSON, with values adapated to their correct Python data types (hstore normally only returns values as strings)
- HstoreToJSONBLoose: Same as HstoreToJSONB, but attempt to distinguish numerical and Boolean values so they are unquoted in the JSON

For more information on these functions, check the postgresql documentation for each one. These functions handle the arguments by converting them to the correct expressions automatically:

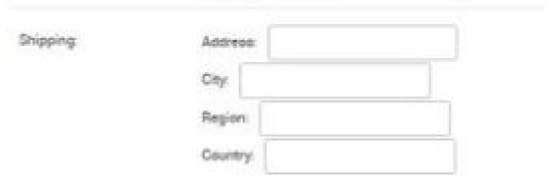
```
from django_postgres_extensions.models.functions import *
obj = Product.objects.queryset.annotate(description_slice=Slice('description', ['Industry', 'Release
obj = Product.objects.update(description = Delete('description', 'Genre'))
obj = Product.objects.update(description = Delete('description', ['Industry', 'Genre']))
Product.objects.update(description=Delete('description', {'Industry': 'Music', 'Release': 'Song', 'Ge
Product.objects.annotate(description_keys=AKeys('description')).get()
Product.objects.annotate(description_values=AVals('description')).get()
Product.objects.annotate(description_array=HStoreToArray('description')).get()
Product.objects.annotate(description_matrix=HStoreToMatrix('description')).get()
Product.objects.annotate(description_jsonb=HstoreToJSONB('description')).get()
Product.objects.annotate(description_jsonb=HstoreToJSONBLoose('description')).get()
```

5.4 Use With Nested Form Field

django.contrib.postgres includes a HStoreField for forms where you have to enter a hstore value programatically. Django Postgres Extensions adds a NestedFormField and NestedFormWidget (subclassed from the Django MultiValue Field and Widget) for use with a HStore Field. To use it specify a list of fields or a list of keys as a keyword argument to the Hstore Model field, but not both:

```
class Product(models.Model):
    shipping = HStoreField(keys=('Address', 'City', 'Region', 'Country'), blank=True, default={})
```

The field would look like:



CHAPTER 6

JSONField

6.1 Basic Usage

To use the JSON field:

```
from django.db import models
from django_postgres_extensions.models.fields import JSONField

class Product(models.Model):
    description = JSONField(null=True, blank=True)
```

6.2 Individual keys

- Get json values by key or key path:

```
from django_postgres_extensions.models.expressions import Key
obj = Product.objectsannotate(Key('description', 'Details')).get()
obj = Product.objects.annotate(Key('description', 'Details__Rating')).get()
obj = Product.objects.annotate(Key('description', 'Tags__1')).get()
```

- Update JSON Field by specific keys, leaving any others untouched:

```
Product.objects.update(description__ = {'Industry': 'Movie', 'Popularity': 'Very Popular'})
```

- Delete JSONField by key or key path:

```
Product.objects.update(description__del ='Details')
Product.objects.update(description__del = 'Details__Release')
Product.objects.update(description__del='Tags__1')
```

6.3 Database functions

Various database functions are included for interacting with JSONFields:

- JSONBSet: updates individual keys in the JSONField without modifying the others.
- JSONBArrayLength: returns the length of a JSONField who's parent object is an array.

Check the postgresql documentation for more information on these functions. These functions handle the arguments by converting them to the correct expressions automatically:

```
from django_postgres_extensions.models.functions import *
from psycopg2.extras import Json
obj = Product.objects.update(description = JSONBSet('description', ['Details', 'Genre'], Json('Heavy
obj = Product.objects.update(description = JSONBSet('description', ['1', 'c'], Json('g')))
obj = Product.objects.queryset.annotate(tags_length=JSONBArrayLength('tags', 1)).get()
```

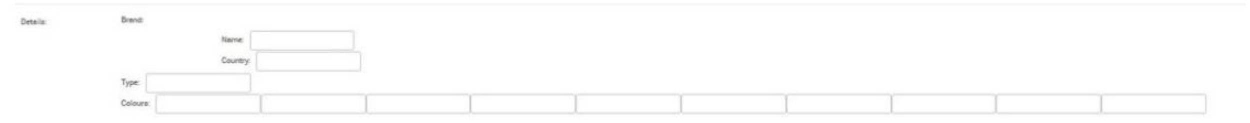
6.4 Use With NestedFormField

The same NestedFormField and NestedFormWidget referred in the HStore description can also be used with a JSON Field. To use it give the fields keyword argument:

```
details_fields = (
    NestedFormField(label='Brand', keys=('Name', 'Country')),
    forms.CharField(label='Type', max_length=25, required=False),
    SplitArrayField(label='Colours', base_field=forms.CharField(max_length=10, required=False), size=
)

class Product(models.Model):
   details = JSONField(fields=details_fields, blank=True, default={})
```

The field would look like:



CHAPTER 7

Array Many To Many Field

7.1 Basic Usage

The Array Many To Many Field is designed be a drop-in replacement for the normal Django Many To Many Field except that it uses an array instead of a separate table to store relationships, but replicates many of the same features. In general, write queries are much faster than the traditional M2M however select queries are typically slower.

To use this field, it is required to set ENABLE_ARRAY_M2M = True in settings.py (to enable the required monkey-patching):

```
ENABLE_ARRAY_M2M = True
```

Then in models.py:

```
from django.db import models
from django_postgres_extensions.models.fields import ArrayManyToManyField

class Publication(models.Model):
    title = models.CharField(max_length=30)

    def __str__(self):
        return self.title

    class Meta:
        ordering = ('title',)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = ArrayManyToManyField(Publication, name='publications')

    def __str__(self):
        return self.headline

    class Meta:
        ordering = ('headline',)
```

The Array Many To Many field supports the following features which replicate the API of the regular Many To Many Field:

- Descriptor queryset with add, remove, clear and set for both forward and reverse relationships
- Prefetch related for both forward and reverse relationships
- Lookups across relationships with filter for both forward and reverse relationships

- Lookups across relationships with exclude for for forward relationships only

You can find more information on how these features work in the Django documentation for the regular Many To Many Field:

https://docs.djangoproject.com/en/1.9/topics/db/examples/many_to_many/

CHAPTER 8

Querysets

8.1 Additional Queryset Methods

This app adds the format method to all querysets. This will defer a field and add an annotation with a different format. For example to return a hstorefield as json:

```
qs = Model.objects.all().format('description', HstoreToJSONBLoose)
```

CHAPTER 9

Running Tests

9.1 Running

To run the tests

```
$ git clone https://github.com/primal100/django_postgres_extensions.git
dpe_repo
```

```
$ cd dpe_repo/tests
```

Configure the postgresql connection details in test_postgres.py.

```
$ ./runtests.py --exclude-tag=benchmark
```

9.2 Benchmarks

Benchmark tests are included to compare performance of the Array M2M with the traditional Django table-based M2M. They can be quite slow and thus it is recommended to exclude them when running tests altogether as in the above example.

They can be run with:

```
$ ./runtests.py benchmarks.tests
```

CHAPTER 10

User Guide

Getting started describes how to get up and running with Django Postgres Extensions. Feature Overview gives a basic overview of the features in Django Postgres Extensions.