django-pgfields Documentation

Release 1.2.0

Luke Sneeringer

January 27, 2014

Contents

This is django-pgfields, a pluggable Django application that adds support for several specialized PostgreSQL fields within the Django ORM.

django-pgfields will work with Django applications using the PostgreSQL backend, and adds support for:

- Arrays
- Composte Types
- JSON
- UUIDs

CHAPTER 1

Dependencies & Limitations

django-pgfields depends on:

- Python 2.7+ or 3.3+ (Python 2.6 probably works, but is not explicitly tested against.)
- Django 1.5+
- Psycopg2 2.5+
- six 1.4.1+

CHAPTER 2

Quick Start

In order to use django-pgfields in a project:

- **Installation**
  - `pip install django-pgfields`
  - Add `django_pg` to your `settings.INSTALLED_APPS`.
- **Usage**
  - Essentially: Import our models module instead of the stock Django module. So, replace `from django.db import models` with `from django_pg import models`.
  - The new field classes provided by `django_pg` are now available on the models module. Use, for instance, `models.UUIDField` and `models.ArrayField` just as you would use `models.CharField`.

CHAPTER 3

Getting Help

If you think you've found a bug in django-pgfields itself, please post an issue on the Issue Tracker.

For usage help, you're free to e-mail the author, who will provide help (on a best effort basis) if possible.

CHAPTER 4

License

New BSD.

CHAPTER 5

Index

5.1 Using django-pgfields

Using django-pgfields' extension to the Django ORM is fairly straightforward. You don't need to use a custom backend (indeed, django-pgfields does not provide one).

5.1.1 Instructions

The short version of usage is that in order to use the features that django-pgfields adds, you need to do two things.

Add django_pg to INSTALLED_APPS

First, you must add `django_pg` to your settings module's `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # your other apps
    'django_pg',
]
```

It doesn't matter where in the list you add it, as long as it's present.

from django_pg import models

Second, import django-pgfields' `models` module instead of the one supplied by Django.

So, everywhere that you would write this:

```
from django.db import models
```

Instead, write this:

```
from django_pg import models
```

Internally, django-pgfields loads all of the things provided by the Django `models` module, subclassing certain items needed to make everything work, and adding the fields it provides.

5.1.2 Explanation

Django provides a rich ORM with a valuable and customizable QuerySet API. One aspect of this ORM is a high wall of separation between the use of data in your application (such as the Python objects that are assigned to model instance attributes) and the actual SQL that is generated to perform operations, which additionally also changes to account for the fact that Django ships with four backends (and several more are available).

One consequence of this design is that Field subclasses have a somewhat restricted set of overridable bits. In particular, they can't (easily) touch the representation of database field names or operators. This is delegated to the backend and to a series of specialized classes which are responsible for generating various pieces of the final SQL query.

The ultimate choreographer of this complex dance is the Manager class. The `Manager` class instantiates the `QuerySet` class, which in turn instantiates internal classes such as `WhereNode` and `SQLExpression` which are ultimately responsible for taking your querysets and constructing actual queries suitable for your backend. Field classes have a very defined (and limited) role in this dance, to avoid breaking down the wall between the different segments of logic.

Complex fields like `ArrayField` and `CompositeField` are non-trivial, and aren't use cases covered by Django's stock query construction classes. Therefore, in order for them to function correctly, these classes must be subclassed.

Importing your models module from django_pg instead of from django.db means that you get django-pgfields' subclasses of `Model` and `Manager` which enable this extra functionality, as well as providing additional (optional) hooks for other Field subclasses.

5.2 Simple Fields

django-pgfields exposes several new fields corresponding to data types available in PostgreSQL that are not available in other databases supported by Django.

These fields are available on the `django_pg.models` module (see *Using django-pgfields* for more on this).

5.2.1 Array Field

PostgreSQL supports an array datatype. This is most similar to arrays in many statically-typed languages such as C or Java, in that you explicitly declare that you want an array of a specific type (for instance, an array of integers or an array of strings).

django-pgfields exposes this by having the array field accept another field as its initial argument (or, alternatively, by using the `of` keyword argument).:

```
from django_pg import models

class Hobbit(models.Model):
    name = models.CharField(max_length=50)
    favorite_foods = models.ArrayField(models.CharField(max_length=100))
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
```

This will create an array of strings in the database (to be precise: `character varying(100) []`). Assignment of values is done using standard Python lists:

```
pippin = Hobbit.objects.create(
    name='Peregrin Took',
    favorite_foods=['apples', 'lembas bread', 'potatoes'],
)
```

As a note, do not attempt to store a full list of any hobbit's favorite foods. Your database server does not have sufficient memory or swap space for such a list.

Lookups

When looking up data against an array field, the field supports three lookup types: `exact` (implied), `contains`, and `len`.

exact

The `exact` lookup type is the implied lookup type when doing a lookup in the Django ORM, and does not need to be explicitly specified. A straight lookup simply checks for array equality. Continuing the example immediately above:

```
>>> hobbit = Hobbit.objects.get(
    favorite_foods=['apples', 'lembas bread', 'potatoes'],
)
>>> hobbit.name
'Peregrin Took'
```

contains

The `contains` lookup type checks to see whether *all* of the provided values exist in the array. If you only need to check for a single value, and the value is not itself an array (in a nested case, for instance), you may specify the lookup value directly:

```
>>> hobbit = Hobbit.objects.get(favorite_foods__contains='apples')
>>> hobbit.name
'Peregrin Took'
```

If you choose to do a `contains` lookup on multiple values, then be aware that *order is not relevant*. The database will check to ensure that each value is present, but ignore order of values in the array altogether:

```
>>> hobbit = Hobbit.objects.get(
    favorite_foods__contains=['lembas bread', 'apples'],
)
>>> hobbit.name
'Peregrin Took'
```

len

The `len` lookup type checks the *length of* the array, rather than its contents. It maps to the array_length function in PostgreSQL (with the second argument set to `1`).

Such lookups are simple and straightforward:

```
>>> hobbit = Hobbit.objects.get(favorite_foods__len=3)
>>> hobbit.name
'Peregrin Took'
```

5.2.2 JSON Field

PostgreSQL 9.2 added initial support for a JSON data type. If you wish to store JSON natively in PostgreSQL, use the JSONField field:

```
from django_pg import models

class Dwarf(models.Model):
    name = models.CharField(max_length=50)
    data = models.JSONField()
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
```

If you're using a version of PostgreSQL earlier than 9.2, this field will fall back to the `text` data type.

Warning: As of PostgreSQL 9.2, *storing* JSON is fully supported, but doing any useful kind of lookup (including direct equality) on it is not.
As such, django-pgfields supports storing JSON data, and will return the JSON fields' data to you when you lookup a record by other means, but it does *not* support *any* kind of lookup against JSON fields. Attempting *any* lookup will raise TypeError.

Values

The JSON field will return values back to you in the Python equivalents of the native JavaScript types:

- JavaScript `number` instances will be converted to `int` or `float` as appropriate.
- JavaScript `array` instances will be converted to Python `list` instances, and value conversion will be recursively applied to every item in the list.
- JavaScript `object` instances will be converted to Python `dict`, and value conversion will be recursively applied to the keys and values of the dictionary.
- JavaScript `string` instances will be converted to Python 3 `str`.
- JavaScript `boolean` instances will be converted to Python `bool`.
- JavaScript `null` is converted to Python `None`.
- JavaScript special values (`NaN`, `Infinity`) are converted to their Python equivalents. Use `math.isnan` and `math.isinf` to test for them.

Note: Because field subclasses are called to convert values over and over again, there are a few cases where the conversion is not idempotent. In particular, strings that are also valid JSON (or look sufficiently close to valid JSON) will be deserialized again.

The short version: write Python dictionaries, lists, and scalars, and the JSON field will figure out what to do with it.

5.2.3 UUID Field

In order to store UUIDs in the database under the PostgreSQL UUID type, use the UUIDField field:

```
from django_pg import models

class Elf(models.Model):
    id = models.UUIDField(auto_add=True, primary_key=True)
    name = models.CharField(max_length=50)
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
```

Options

The UUID field implements the following field options in addition to the field options available to all fields.

Note: The UUID field interprets and writes blank values as SQL `NULL`. Therefore, setting `blank=True` requires `null=True` also. Setting the former but not the latter will raise `AttributeError`.

auto_add

Normally, the UUIDField works like any other Field subclass; you are expected to provide a value, and the value is saved to the database directly.

If `auto_add=True` is set, then explicitly providing a value becomes optional. If no value is provided, then the field will auto-generate a random version 4 UUID, which will be saved to the database (and assigned to the model instance).

This is a particularly useful construct if you wish to store UUIDs for primary keys; they're a completely acceptable substitute for auto-incrementing integers:

```
>>> legolas = Elf(name='Legolas Greenleaf')
>>> legolas.id
''
>>> legolas.save()
>>> legolas.id
UUID('b1f12115-3337-4ec0-acb9-1bcf63e44477')
```

coerce_to

By default, the `to_python` method on `UUIDField` will coerce values to UUID objects. Setting this option will use a different class constructor within `to_python`.

The general use-case for this is if you want to get strings instead of UUID objects. The following example would be the output in the case that you assigned `coerce_to=str`:

```
>>> legolas = Elf(name='Legolas Greenleaf')
>>> legolas.save()
>>> legolas.id
'b1f12115-3337-4ec0-acb9-1bcf63e44477'
```

Values

The UUID field will return values from the database as Python UUID objects.

If you choose to do so, you may assign a valid string to the field. The string will be converted to a `uuid.UUID` object upon assignment to the instance:

```
>>> legolas = Elf(name='Legolas Greenleaf')
>>> legolas.id = '01234567-abcd-abcd-abcd-0123456789ab'
>>> legolas.id
UUID('01234567-abcd-abcd-abcd-0123456789ab')
>>> type(legolas.id)
<class 'uuid.UUID'>
```

Lookups can be performed using either strings or Python UUID objects.

5.3 Composite Fields

In addition to a generous set of built-in field types, PostgreSQL allows for the definition of custom fields on a per-schema level. Composite fields are simply custom fields that are composites of an ordered list of key names and field types already provided by PostgreSQL.

Composite fields come with a few limitations compared to standard tables. They can't have constraints of any kind, making them a poor choice for anything requiring a foreign key. Similarly, if you're doing lookups based on a composite field, you should know precisely what you're doing.

If you aren't familiar with PostgreSQL composite fields and want to understand more about them, you should consult the PostgreSQL composite fields documentation before continuing on.

5.3.1 Defining Composite Fields in the ORM

The representation of composite fields in the ORM using django-pgfields should be remarkably similar to the representation of models themselves, since they're conceptually quite similar.

Differences from Model subclasses

A few things differ between models and composite fields:

- Composite fields inherit from `django_pg.models.CompositeField` rather than `django.db.models.Model`.
- Composite fields do not get an `id` field by default, and do not need one.
- Composite fields may not contain any subclass of `django.db.models.RelatedField`. This includes `ForeignKey`, `OneToOneField`, or `ManyToManyField` fields.
- Any constraints provided to composite fields will be ignored at the database level.
  - Exception: `max_length` sent to `CharField`. This is part of the type definition, and is still required.
- Most `Meta` options no longer have any meaning, and a new `Meta` option (`db_type`) is available to composite fields.
- Composite fields can't do lookups based on a single key in the composite field. PostgreSQL has this ability, but it's not yet implemented in django-pgfields.

Type Definition Example

With these differences in mind, creating a composite field is straightfoward and familiar:

```
from django_pg import models

class AuthorField(models.CompositeField):
    name = models.CharField(max_length=75)
    sex = models.CharField(max_length=6, choices=(
        ('male', 'Male'),
        ('female', 'Female'),
    ))
    birthdate = models.DateField()
```

Once the subclass is defined, it can be used within a model like any other field:

```
class Book(models.Model):
    title = models.CharField(max_length=50)
    author = AuthorField()
    date_published = models.DateField()
```

Meta Options

db_type

All types in PostgreSQL have a name to identify them, such as `text` or `int`. Your custom type must also have a name.

If you don't provide one, django-pgfields will introspect it from the name of the class, by converting the class name to lower-case, and then stripping off `"field"` from the end if it's present. So, in the example above, our `AuthorField` would create an `author` type in the schema.

You may choose to provide one by specifying `db_type` in the field's inner `Meta` class:

```
class AuthorField(models.CompositeField):
    name = models.CharField(max_length=75)
    sex = models.CharField(max_length=6, choices=(
        ('male', 'Male'),
        ('female', 'Female'),
    ))
    birthdate = models.DateField()

    class Meta:
        db_type = 'ns_author'
```

Manual specification of the composite type's name is recommended, if only so that they're namespaced (to a degree). You don't want your type name to conflict with some new type that PostgreSQL may add in the future, after all.

5.3.2 Assigning Values to Composite Fields

The presence of any composite field entails the need to write data to the model instance containing that field. There are two ways to go about this: by using a tuple, or by using a special "instance class" created when you instantiate the field subclass.

Tuples

In many simple circumstances, the quickest way to assign values is to use a tuple. PostgreSQL accepts its write values to composite fields in a tuple-like structure, with values provided in a specified order (the order of the fields) and keys omitted.

This is a legal way to assign an author to a book:

```
>>> hobbit = Book(title='The Hobbit', date_published=date(1937, 9, 21))
>>> hobbit.author = ('J.R.R. Tolkien', 'male', date(1892, 1, 3))
>>> hobbit.save()
```

Composite Instances

The above method works fine in simple cases, but isn't great for more complex ones, especially since tuples are immutable. Fortunately, there's a solution. Whenever a composite field is created, a "composite instance" class is created alongside of it, and is available under the `instance_class` property of the field.

This example is identical in function to the tuple example shown above:

```
>>> hobbit = Book(title='The Hobbit', date_published=date(1937, 9, 21))
>>> hobbit.author = AuthorField.instance_class(
    birthdate=date(1892, 1, 3),
    name='J.R.R. Tolkien',
    sex='male',
)
>>> hobbit.save()
```

The actual name of the instance class is derived from the name of the field, by dropping the name `Field` (if present) from the field name's subclass. If the instance name does not conflict with the field name, it is automatically assigned to the same module in which the instance was created.

In the above example, assuming that `AuthorField` was defined in the `library.models` module, we'd be able to do this:

```
>>> from library.models import Book, Author
>>> hobbit = Book(title='The Hobbit', date_published=date(1937, 9, 21))
>>> hobbit.author = Author(
    birthdate=date(1892, 1, 3),
    name='J.R.R. Tolkien',
    sex='male',
)
>>> hobbit.save()
```

5.3.3 Accessing Composite Values

When values are being *read*, a composite instance is always used, never a tuple. If a tuple is required, it can be explicitly typecast.

Composite values access *their* individual fields as attributes, just like subclasses of Model:

```
>>> hobbit = Book.objects.get(title='The Hobbit')
>>> hobbit.author.name
'J.R.R. Tolkien'
>>> hobbit.author.birthdate
date(1892, 1, 3)
```

5.4 Miscellaneous

Miscellaneous features provided by django-pgfields that are not actually PostgreSQL-related fields.

5.4.1 Improved Repr

django-pgfields adds an optional, opt-in improved `__repr__` method on the base Model class.

The default `__repr__` implementation on the Model class simply identifies the model class to which the instance belongs, and does nothing else:

```
>>> mymodel = MyModel.objects.create(spam='eggs', foo='bar')
>>> mymodel
<MyModel: MyModel object>
```

The improved `__repr__` implementation that django-pgfields provides iterates over the fields on the model and prints out a readable structure:

```
>>> mymodel = MyModel.objects.create(spam='eggs', foo='bar')
>>> mymodel
<MyModel: { 'id': 1, 'spam': 'eggs', 'foo': 'bar' }>
```

This is more useful for debugging, logging, and working on the shell.

Settings

django-pgfields exposes this functionality through optional settings in your Django project.

DJANGOPG_IMPROVED_REPR

- default: `False`

Set this to `True` to enable the improved repr. Because providing an alternate `__repr__` implementaiton is not the core function of django-pgfields, it is offered on an opt-in basis.

DJANGOPG_REPR_TEMPLATE

- default: `'single_line'`

django-pgfields offers two built-in templates for printing model objects: a single-line template and a multi-line template. They are the same, except the model-line template adds line breaks and indentation for increased readability. However, this readability may come at the expense of ease of parsing logs.

- Set this to `'single_line'` for the single-line template (default).
- Set this to `'multi_line'` for the multi-line template.

The single-line template produces output like this:

```
>>> mymodel = MyModel.objects.create(spam='eggs', foo='bar')
>>> mymodel
<MyModel: { 'id': 1, 'spam': 'eggs', 'foo': 'bar' }>
```

The multi-line template produces output like this:

```
>>> mymodel = MyModel.objects.create(spam='eggs', foo='bar')
>>> mymodel
<MyModel: {
    'id': 1,
    'spam': 'eggs',
    'foo': 'bar'
}>
```

Additionally, you may define your own template by providing a two-tuple to this setting. Each tuple should be a string. The first string is the overall template, and the second string is the glue on which the individual fields are joined.

The template is populated using the % operator, and it is passed a dictionary with four elements:

- `class_name`: The name of the model class
- `members`: The model's members, joined on the join glue
- `tab`: The appropriate tab depth
- `untab`: The appropriate tab depth for a depth one above; useful for closing off a structure

The glue is sent only the `tab` variable.

5.4.2 Improved select_related and prefetch_related

django-pgfields adds `select_related` and `prefetch_related` as options that can be specified in the `Meta` inner class on a model. This allows a developer to gain the database efficiency of these methods if he knows that he will always want this or that related object.

The syntax is thus:

```
from django_pg import models

class MyModel(models.Model):
    my_other_model = models.ForeignKey(MyOtherModel)

    class Meta:
        select_related = 'my_other_model'
```

If more than one field should be included, `select_related` can be specified as an iterable:

```
class Meta:
    select_related = ('foo', 'bar')
```

This will automatically cause querysets returned from `Manager.get_queryset` to apply the appropriate `select_related` call.

Note that while all of the above examples use `select_related`, this same syntax also works for `prefetch_related`.

5.5 Settings

django-pgfields provides several settings, which will customize its operation.

5.5.1 DJANGOPG_CUSTOM_MANAGER

- default: `None`

django-pgfields 1.0 builds on previous versions of django-pgfields by providing *two* Manager classes, one which subclasses the vanilla Django Manager, and another which subclasses the GeoManager provided with Django's GIS application, GeoDjango.

django-pgfields will automatically introspect which of these to use by looking at the backend of the default database in your database settings.

However, if that result isn't what you want, or if you want a custom manager to be applied across-the-board to all models that subclass `django_pg.models.Model`, then set this to the particular Manager subclass that you'd like.

You can do this by either providing the full module and class path as a string, or by providing the class directly.

5.5.2 DJANGOPG_IMPROVED_REPR

- default: `False`

Set this to `True` to enable the improved repr. Because providing an alternate `__repr__` implementaiton is not the core function of django-pgfields, it is offered on an opt-in basis.

See the improved repr documentation for more details.

5.5.3 DJANGOPG_REPR_TEMPLATE

- default: `'single_line'`

Sets the template that is used by the improved repr provided by django-pgfields. See the improved repr documentation for more details.

5.5.4 DJANGOPG_DEFAULT_UUID_PK

- default: `False`

If set to `True`, this will cause models to get a UUID as their default primary key if none is specified, rather than an auto-incrementing integer.

Note that this does not currently work on `ManyToManyField` instances that are automatically generated, as they inherit from `django.db.models.Model`.

5.6 Releases

This page contains release notes for django-pgfields. Consult the release notes to read about new features added in each release, and to be aware of any backwards incompatible changes.

5.6.1 django-pgfields 1.3

Welcome to django-pgfields 1.3!

Overview

This release builds on django-pgfields by adding support for automatically generated UUID primary keys on most models, as well as a more convenient way to specify `select_related` and `prefetch_related`.

Features

- django-pgfields now defines a `DJANGOPG_DEFAULT_UUID_PK` setting. If set to `True` (the default is `False`), it will cause most models created with no explicitly-specified primary key to get a UUID primary key field, rather than an auto-incrementing primary key.
- django-pgfields now builds on Django's default `Model` by adding support for `select_related` and `prefetch_related` to be specified as options on the `Meta` inner class.

5.6.2 django-pgfields 1.2

Welcome to django-pgfields 1.2!

Overview

This release builds on django-pgfields 1.1 by adding support for coercing UUID fields to classes other than uuid.UUID.

Features

- UUIDField now supports a `coerce_to` argument, defaulting to `uuid.UUID`. The obvious use case for setting this is if you need to get back str objects instead.

Bugfixes

- django-pgfields 1.2.1 fixes a bug regarding Unicode values within `CompositeField` subclasses; sending unicode values will now work.

5.6.3 django-pgfields 1.1

Welcome to django-pgfields 1.1!

Overview

This release builds on django-pgfields 1.0 by adding support for Python 2. django-pgfields now adds six as a dependency, and is fully tested against Python 2.7.

django-pgfields is not tested against Python 2.6 (but it's likely that it will work under Python 2.6).

Features

- Python 2.7 support.

5.6.4 django-pgfields 1.0

Welcome to django-pgfields 1.0!

Overview

This release is considered the first true, stable release of django-pgfields. It adds sufficient support for tools within the Django ecosystem (gis, south) to be usable, and all documented aspects of the module should not change at this point, except to accomodate a substantial change in Django itself.

This release of django-pgfields contains the following major features:

Features

- Added GeoDjango support.
- Added an improved repr implementation to Model.

GeoDjango Support

django-pgfields 1.0 resolves a previously-existing conflict between django-pgfields and GeoDjango (`django.contrib.gis`).

Because both django-pgfields and GeoDjango implement their additional features by subclassing several classes within Django itself, it wasn't possible to use both of them.

django-pgfields 1.0 removes this limitation by introspecting your database backend setting and assigning to models an appropriate Manager class. So, if you are using the `django.contrib.gis.db.backends.postgis` backend, you'll get our GeoManager subclass (without having to specify anything!), while if you're using the vanilla Django PostgreSQL backend, you'll get our Manager subclass.

This functionality is provided across-the-board. However, if our introspection gets it wrong, you can define a `DJANGOPG_DEFAULT_MANAGER` setting, which will be applied to all models in lieu of django-pgfields' introspection. This is easier than manually assigning a manager to every model.

Improved Repr

django-pgfields 1.0 provides an improved `__repr__` implementation on the base Model, from which your models inherit.

As this is not the core mission of django-pgfields, this functionality is provided on an opt-in basis, and will only be activated if you set a particular setting.

For more information, see the improved repr documentation.

Minor Improvements

- If `auto_add=True` is set on a UUID field, it now implies `editable=False`. This makes it behave as much like AutoField as possible, since a primary key replacement is one of the most likely reasons to use `auto_add` at all.
- When using an `ArrayField`, any item that is added to the list is automatically coerced to the sub-field's format.
- When a value for a `CompositeField` subclass is being coerced to the appropriate Python type, all sub-items are coerced as well.
- When using a `CompositeField`, any item that is added to a key of the field is automatically coerced to that key's format.

5.6.5 django-pgfields 0.9.2

Welcome to django-pgfields 0.9.2!

Overview

This release of django-pgfields contains the following major features:

- South Support
- JSON Field

Features

South Support

django-pgfields 0.9.2 adds out-of-the-box support for South, the popular migrations application for Django. With this release, all fields provided by django-pgfields are given appropriate introspection rules for South.

This also applies to any subclasses of `CompositeField`, which are given automatic introspection rules. This does *not* examine your subclass, however, so it is your responsibility to add introspection rules if your subclass takes constructor arguments.

Note: South migrations using ArrayField were broken in django-pgfields 0.9.2; while the initial migration including the ArrayField would run, subsequent migrations would not. This has been corrected and a subsequent release, django-pgfields 0.9.2.1, issued.

JSON Field

django-pgfields 0.9.2 adds a new field for storing JSON in the database. The new JSONField is capable of taking Python objects that serialize into JSON (such as lists, dicts, booleans, and strings) and storing the JSON representation.

This feature makes use of the new JSON data type introduced in PostgreSQL 9.2. On earlier versions of PostgreSQL, it will fall back to the text type.

More information is available on the *Fields* documentation page.