djangopostgres Documentation

Release 0.0.1

Author

May 16, 2017

Contents

Adds first-class support for PostgreSQL features to the Django ORM.

Planned features include:

- Arrays
- Enums
- Bit Strings
- Constraints
- Triggers
- Domains
- Composite Types
- Views

Obviously this is quite a large project, but I think it would provide a huge amount of value to Django developers.

CHAPTER 1

Why?

PostgreSQL is an excellent data store, with a host of useful and efficiently-implemented features. Unfortunately these features are not exposed through Django's ORM, primarily because the framework has to support several SQL backends and so can only provide a set of features common to all of them.

The features made available here replace some of the following practices:

- Manual denormalization on `save()` (such that model saves may result in three or more separate queries).
- Sequences represented by a one-to-many, with an `order` integer field.
- Complex types represented by JSON in a text field.

CHAPTER 2

Contents

This is a WIP, so the following list may grow and change over time.

Views

For more info on Postgres views, see the official Postgres docs. Effectively, views are named queries which can be accessed as if they were regular database tables.

Quickstart

Given the following view in SQL:

```
CREATE OR REPLACE VIEW myapp_viewname AS
SELECT * FROM myapp_table WHERE condition;
```

You can create this view by just subclassing *django_postgres.View*. In `myapp/models.py`:

```
import django_postgres

class ViewName(django_postgres.View):
    projection = ['myapp.Table.*']
    sql = """SELECT * FROM myapp_table WHERE condition"""
```

View

class `django_postgres.`**View**

Inherit from this class to define and interact with your database views.

You need to either define the field types manually (using standard Django model fields), or use *projection* to copy field definitions from other models.

sql
The SQL for this view (typically a `SELECT` query). This attribute is optional, but if present, the view will be created on `sync_pgviews` (which is probably what you want).

projection
A list of field specifiers which will be automatically copied to this view. If your view directly presents fields from another table, you can effectively 'import' those here, like so:

```
projection = ['auth.User.username', 'auth.User.password',
              'admin.LogEntry.change_message']
```

If your view represents a subset of rows in another table (but the same columns), you might want to import all the fields from that table, like so:

```
projection = ['myapp.Table.*']
```

Of course you can mix wildcards with normal field specifiers:

```
projection = ['myapp.Table.*', 'auth.User.username', 'auth.User.email']
```

Primary Keys

Django requires exactly one field on any relation (view, table, etc.) to be a primary key. By default it will add an `id` field to your view, and this will work fine if you're using a wildcard projection from another model. If not, you should do one of three things. Project an `id` field from a model with a one-to-one relationship:

```
class SimpleUser(django_postgres.View):
    projection = ['auth.User.id', 'auth.User.username', 'auth.User.password']
    sql = """SELECT id, username, password, FROM auth_user;"""
```

Explicitly define a field on your view with `primary_key=True`:

```
class SimpleUser(django_postgres.View):
    projection = ['auth.User.password']
    sql = """SELECT username, password, FROM auth_user;"""
    # max_length doesn't matter here, but Django needs something.
    username = models.CharField(max_length=1, primary_key=True)
```

Or add an `id` column to your view's SQL query (this example uses window functions):

```
class SimpleUser(django_postgres.View):
    projection = ['auth.User.username', 'auth.User.password']
    sql = """SELECT username, password, row_number() OVER () AS id
             FROM auth_user;"""
```

Creating the Views

Creating the views is simple. Just run the `sync_pgviews` command:

```
$ ./manage.py sync_pgviews
Creating views for django.contrib.auth.models
Creating views for django.contrib.contenttypes.models
Creating views for myapp.models
myapp.models.Superusers (myapp_superusers): created
```

```
myapp.models.SimpleUser (myapp_simpleuser): created
myapp.models.Staffness (myapp_staffness): created
```

Migrations

Views play well with South migrations. If a migration modifies the underlying table(s) that a view depends on so as to break the view, that view will be silently deleted by Postgres. For this reason, it's important to run `sync_pgviews` after `migrate` to ensure any required tables have been created/updated.

Bit Strings

Postgres has a bit string type, which is exposed by django-postgres as *`BitStringField`* and the *`BitStringExpression`* helper (aliased as `django_postgres.B`). The representation of bit strings in Python is handled by the python-bitstring library (a dependency of `django-postgres`).

Quickstart

Given the following `models.py`:

```
from django.db import models
import django_postgres

class BloomFilter(models.Model):
    name = models.CharField(max_length=100)
    bitmap = django_postgres.BitStringField(max_length=8)
```

You can create objects with bit strings, and update them like so:

```
>>> from django_postgres import Bits
>>> from models import BloomFilter

>>> bloom = BloomFilter.objects.create(name='test')
INSERT INTO myapp_bloomfilter
  (name, bitmap) VALUES ('test', B'00000000')
  RETURNING myapp_bloomfilter.id;

>>> print bloom.bitmap
Bits('0x00')
>>> bloom.bitmap |= Bits(bin='00100000')
>>> print bloom.bitmap
Bits('0x20')

>>> bloom.save(force_update=True)
UPDATE myapp_bloomfilter SET bitmap = B'00100000'
 WHERE myapp_bloomfilter.id = 1;
```

Several query lookups are defined for filtering on bit strings. Standard equality:

```
>>> BloomFilter.objects.filter(bitmap='00100000')
SELECT * FROM myapp_bloomfilter WHERE bitmap = B'00100000';
```

You can also test against bitwise comparison operators (`and`, `or` and `xor`). The SQL produced is slightly convoluted, due to the few functions provided by Postgres:

```
>>> BloomFilter.objects.filter(bitmap__and='00010000')
SELECT * FROM myapp_bloomfilter WHERE position(B'1' IN bitmap & B'00010000') > 0
>>> BloomFilter.objects.filter(bitmap__or='00010000')
SELECT * FROM myapp_bloomfilter WHERE position(B'1' IN bitmap | B'00010000') > 0
>>> BloomFilter.objects.filter(bitmap__xor='00010000')
SELECT * FROM myapp_bloomfilter WHERE position(B'1' IN bitmap # B'00010000') > 0
```

Finally, you can also test the zero-ness of left- and right-shifted bit strings:

```
>>> BloomFilter.objects.filter(bitmap__lshift=3)
SELECT * FROM myapp_bloomfilter WHERE position(B'1' IN bitmap << 3) > 0
>>> BloomFilter.objects.filter(bitmap__rshift=3)
SELECT * FROM myapp_bloomfilter WHERE position(B'1' IN bitmap >> 3) > 0
```

Bit String Fields

class `django_postgres.`**`BitStringField`**(*max_length=1*[, *varying=False*, ...])

A bit string field, represented by the Postgres `BIT` or `VARBIT` types.

Parameters

- **`max_length`** – The length (in bits) of this field.
- **`varying`** – Use a `VARBIT` instead of `BIT`. Not recommended; it may cause strange querying behavior or length mismatch errors.

If `varying` is True and `max_length` is `None`, a `VARBIT` of unlimited length will be created.

The default value of a `BitStringField` is chosen as follows:

- If a `default` kwarg is provided, that value is used.
- Otherwise, if `null=True`, the default value is `None`.
- Otherwise, if the field is not a `VARBIT`, it defaults to an all-zero bit string of `max_length` (remember, the default length is 1).
- Finally, all other cases will default to a single `0`.

All other parameters (`db_column`, `help_text`, etc.) behave as standard for a Django field.

Bit String Expressions

It's useful to be able to atomically modify bit strings in the database, in a manner similar to Django's F-expressions. For this reason, *`BitStringExpression`* is provided, and aliased as `django_postgres.B` for convenience.

Here's a short example:

```
>>> from django_postgres import B
>>> BloomFilter.objects.filter(id=1).update(bitmap=B('bitmap') | '00001000')
UPDATE myapp_bloomfilter SET bitmap = bitmap | B'00001000'
 WHERE myapp_bloomfilter.id = 1;
>>> bloom = BloomFilter.objects.get(id=1)
>>> print bloom.bitmap
Bits('0x28')
```

class `django_postgres.`**`BitStringExpression`**(*field_name*)

The following operators are supported:

- Concatenation (`+`)
- Bitwise AND (`&`)
- Bitwise OR (`|`)
- Bitwise XOR (`^`)
- (Unary) bitwise NOT (`~`)
- Bitwise left-shift (`<<`)
- Bitwise right-shift (`>>`)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index