django-postgres-copy Documentation

Release 0.0.5

Ben Welsh

Oct 09, 2017

Contents

Quickly move comma-delimited data in and out of a Django model using PostgreSQL's COPY command.

CHAPTER 1

Why and what for?

The people who made this library are data journalists. We are often downloading, cleaning and analyzing new data.

That means we write a load of loaders. In the past we did this by looping through each row and saving it to the database using the Django's ORM create method.

```
import csv
from myapp.models import MyModel


data = csv.DictReader(open("./data.csv"))
for row in data:
    MyModel.objects.create(name=row['NAME'], number=row['NUMBER'])
```

That works, but if you have a big file as Django racks up a database query for each row it can take a long time to get all the data in the database.

Lucky for us, PostgreSQL has a built-in tool called COPY that can hammer data in and out the database with one quick query.

This package tries to make using COPY as easy any other database routine supported by Django. It is implemented by a custom model manager.

Here's how it imports a CSV to a database table.

```
from myapp.models import MyModel


MyModel.objects.from_csv(
    "./data.csv",  # The source file
    dict(name='NAME', number='NUMBER')  # A crosswalk of model fields to CSV headers.
)
```

And here's how it exports a database table to a CSV.

```
from myapp.models import MyModel
```

```
MyModel.objects.to_csv("./data.csv")
```

CHAPTER 2

Installation

The package can be installed from the Python Package Index with *pip*.

```
$ pip install django-postgres-copy
```

You will of course have to have Django, PostgreSQL and an adapter between the two (like psycopg2) already installed to put this library to use.

CHAPTER 3

An example

It all starts with a CSV file you'd like to load into your database. This library is intended to be used with large files but here's something simple as an example.

```
NAME,NUMBER,DATE
ben,1,2012-01-01
joe,2,2012-01-02
jane,3,2012-01-03
```

A Django model that corresponds to the data might look something like this. It should have our custom manager attached.

```
from django.db import models
from postgres_copy import CopyManager


class Person(models.Model):
    name = models.CharField(max_length=500)
    number = models.IntegerField(null=True)
    dt = models.DateField(null=True)
    objects = CopyManager()
```

If the model hasn't been created in your database, that needs to happen.

```
$ python manage.py migrate
```

How to import data

Here's how to create a script to import CSV data into the model. Our favorite way to do this is to write a custom Django management command.

```
from myapp.models import Person
from django.core.management.base import BaseCommand
```

```
class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        Person.objects.from_csv(
            # The path to your CSV
            '/path/to/my/data.csv',
            # And a dict mapping the  model fields to CSV headers
            dict(name='NAME', number='NUMBER', dt='DATE')
        )
```

Run your loader and that's it.

```
$ python manage.py myimportcommand
```

How to export data

```
from myapp.models import Person
from django.core.management.base import BaseCommand


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        # All this method needs is the path to your CSV
        Person.objects.to_csv('/path/to/my/export.csv')
```

Run your exporter and that's it.

```
$ python manage.py myexportcommand
```

That's it. You can even export your queryset after any filters or other tricks. This will work;

```
Person.objects.exclude(name='BEN').to_csv('/path/to/my/export.csv')
```

And so will something like this:

```
Person.objects.annotate(name_count=Count('name')).to_csv('/path/to/my/export.csv')
```

CHAPTER 4

Import options

The `from_csv` manager method has the following arguments and keywords options.

`from_csv`(*csv_path*, *mapping*[, *using=None*, *delimiter=','*, *'*, *null=None*, *force_not_null=None*, *force_null=None*, *encoding=None*, *static_mapping=None*])

Argument	Description
`csv_path`	The path to the delimited data source file (e.g., a CSV)
`mapping`	A dictionary: keys are strings corresponding to the model field, and values correspond to string field names for the CSV header.

Keyword Arguments	
`delimiter`	The character that separates values in the data file. By default it is ",". This must be a single one-byte character.
`quote_character`	Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character.
`null`	Specifies the string that represents a null value. The default is an unquoted empty string. This must be a single one-byte character.
`force_not_null`	Specifies which columns should ignore matches against the null string. Empty values in these columns will remain zero-length strings rather than becoming nulls. The default is None. If passed, this must be list of column names.
`force_null`	Specifies which columns should register matches against the null string, even if it has been quoted. In the default case where the null string is empty, this converts a quoted empty string into NULL. The default is None. If passed, this must be list of column names.
`encoding`	Specifies the character set encoding of the strings in the CSV data source. For example, `'latin-1'`, `'utf-8'`, and `'cp437'` are all valid encoding parameters.
`using`	Sets the database to use when importing data. Default is None, which will use the `'default'` database.
`static_mapping`	Set model attributes not in the CSV the same for every row in the database by providing a dictionary with the name of the columns as keys and the static inputs as values.

Transforming data

By default, the COPY command cannot transform data on-the-fly as it is loaded into the database.

This library first loads the data into a temporary table before inserting all records into the model table. So it is possible to use PostgreSQL's built-in SQL methods to modify values during the insert.

As an example, imagine a CSV that includes a column of yes and no values that you wanted to store in the database as 1 or 0 in an integer field.

```
NAME,VALUE
ben,yes
joe,no
```

A model to store the data as you'd prefer to might look like this.

```
from django.db import models
from postgres_copy import CopyManager


class Person(models.Model):
    name = models.CharField(max_length=500)
    value = models.IntegerField()
    objects = CopyManager()
```

But if the CSV file was loaded directly into the database, you would receive a data type error when the 'yes' and 'no' strings were inserted into the integer field.

This library offers two ways you can transform that data during the insert.

Custom-field transformations

One approach is to create a custom Django field.

You can provide a SQL statement for how to transform the data during the insert into the model table. The transformation must include a string interpolation keyed to "name", where the title of the database column will be slotted.

This example uses a CASE statement to transforms the data.

```
from django.db.models.fields import IntegerField


class MyIntegerField(IntegerField):
    copy_template = """
        CASE
            WHEN "%(name)s" = 'yes' THEN 1
            WHEN "%(name)s" = 'no' THEN 0
        END
    """
```

Back in the models file the custom field can be substituted for the default.

```
from django.db import models
from postgres_copy import CopyManager
from myapp.fields import MyIntegerField


class Person(models.Model):
```

```
    name = models.CharField(max_length=500)
    value = MyIntegerField()
    objects = CopyManager()
```

Run your loader and it should finish fine.

Model-method transformations

A second approach is to provide a SQL string for how to transform a field during the insert on the model itself. This lets you specify different transformations for different fields of the same type.

You must name the method so that the field name is sandwiched between `copy_` and `_template`. It must return a SQL statement with a string interpolation keyed to "name", where the name of the database column will be slotted.

For the example above, the model might be modified to look like this.

```
from django.db import models
from postgres_copy import CopyManager


class Person(models.Model):
    name = models.CharField(max_length=500)
    value = models.IntegerField()
    objects = CopyManager()

    def copy_value_template(self):
      return """
          CASE
              WHEN "%(name)s" = 'yes' THEN 1
              WHEN "%(name)s" = 'no' THEN 0
          END
          """
```

And that's it.

Here's another example of a common issue, transforming the CSV's date format to one PostgreSQL and Django will understand.

```
def copy_mydatefield_template(self):
    return """
        CASE
            WHEN "%(name)s" = '' THEN NULL
            ELSE to_date("%(name)s", 'MM/DD/YYYY') /* The source CSV's date pattern␣
↪can be set here. */
        END
    """
```

It's important to handle empty strings (by converting them to NULL) in this example. PostgreSQL will accept empty strings, but Django won't be able to ingest the field and you'll get a strange "year out of range" error when you call something like `MyModel.objects.all()`.

Inserting static values

If your model has columns that are not in the CSV, you can set static values for what is inserted using the `static_mapping` keyword argument. It will insert the provided values into every row in the database.

An example could be if you want to include the name of the source CSV file along with each row.

Your model might look like this:

```
from django.db import models
from postgres_copy import CopyManager


class Person(models.Model):
    name = models.CharField(max_length=500)
    number = models.IntegerField()
    source_csv = models.CharField(max_length=500)
    objects = CopyManager()
```

And your loader would look like this:

```
from myapp.models import Person
from django.core.management.base import BaseCommand


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        Person.objects.from_csv(
            '/path/to/my/data.csv',
            dict(name='NAME', number='NUMBER'),
            static_mapping = {
                'source_csv': 'data.csv'
            }
        )
```

Extending with hooks

The `from_csv` method connects with a lower level `CopyMapping` class with optional hooks that run before and after the COPY statement. They run first when the CSV is into a temporary table and then again before and after the INSERT statement that then slots data into your model's table.

If you have extra steps or more complicated logic you'd like to work into a loading routine, `CopyMapping` and its hooks provide an opportunity to extend the base library.

To try them out, subclass `CopyMapping` and fill in as many of the optional hook methods below as you need.

```
from postgres_copy import CopyMapping


class HookedCopyMapping(CopyMapping):
    def pre_copy(self, cursor):
        print "pre_copy!"
        # Doing whatever you'd like here

    def post_copy(self, cursor):
        print "post_copy!"
        # And here

    def pre_insert(self, cursor):
        print "pre_insert!"
        # And here
```

```
    def post_insert(self, cursor):
        print "post_insert!"
        # And finally here
```

Now you can run that subclass directly rather than via a manager. The only differences are that model is the first argument `CopyMapping`, which creates an object that is executed with a call to its `save` method.

```
from myapp.models import Person
from myapp.loaders import HookedCopyMapping
from django.core.management.base import BaseCommand


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        # Note that we're using HookedCopyMapping here
        c = HookedCopyMapping(
            Person,
            '/path/to/my/data.csv',
            dict(name='NAME', number='NUMBER'),
        )
        # Then save it.
        c.save()
```

CHAPTER 5

Export options

The `to_csv` manager method only requires one argument, the path to where the CSV should be exported. It also allows users to optionally limit or expand the fields written out by providing them as additional parameters.

to_csv(*csv_path*[, **fields*, *delimiter=', '*])

Argument	Description
`csv_path`	The path to the delimited data source file (e.g., a CSV)
`fields`	Strings corresponding to the model fields to be exported. All fields on the model are exported by default. Fields on related models can be included with Django's double underscore notation.
`delimiter`	String that will be used as a delimiter for the CSV file.

Reducing the exported fields

You can reduce the number of fields exported by providing the ones you want as a list to the `to_csv` method.

Your model might look like this:

```
from django.db import models
from postgres_copy import CopyManager


class Person(models.Model):
    name = models.CharField(max_length=500)
    number = models.IntegerField()
    objects = CopyManager()
```

You could export only the name field by providing it as an extra parameter.

```
from myapp.models import Person
from django.core.management.base import BaseCommand
```

```
class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        Person.objects.to_csv(
            '/path/to/my/export.csv',
            'name'
        )
```

Increasing the exported fields

In cases where your model is connected to other tables with a foreign key, you can increase the number of fields exported to included related tables using Django's double underscore notation.

Your models might look like this:

```
from django.db import models
from postgres_copy import CopyManager


class Hometown(models.Model):
    name = models.CharField(max_length=500)
    objects = CopyManager()


class Person(models.Model):
    name = models.CharField(max_length=500)
    number = models.IntegerField()
    hometown = models.ForeignKey(Hometown)
    objects = CopyManager()
```

You can reach across to related tables during an export by adding their fields to the export method.

```
from myapp.models import Person
from django.core.management.base import BaseCommand


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        Person.objects.to_csv(
            '/path/to/my/export.csv',
            'name',
            'number',
            'hometown__name'
        )
```

CHAPTER 6

Open-source resources

- Code: github.com/california-civic-data-coalition/django-postgres-copy
- Issues: github.com/california-civic-data-coalition/django-postgres-copy/issues
- Packaging: pypi.python.org/pypi/django-postgres-copy
- Testing: travis-ci.org/california-civic-data-coalition/django-postgres-copy
- Coverage: coveralls.io/r/california-civic-data-coalition/django-postgres-copy

Index