django-postgres-copy Documentation

Release 0.0.5

Ben Welsh

Apr 30, 2017

Contents

Quickly load comma-delimited data into 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. You can usually do 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'])
```

But if you have a big CSV, Django will rack up database queries and it can take a long time to finish.

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

This package tries to make using COPY as easy any other database routine supported by Django. It is largely based on the design of the LayerMapping utility for importing geospatial data.

```
from myapp.models import MyModel
from postgres_copy import CopyMapping

c = CopyMapping(
    MyModel,
    "./data.csv",
    dict(name='NAME', number='NUMBER')
)
c.save()
```

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.

```
from django.db import models


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

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

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

Create a loader that uses this library to load CSV data into the model. One place you could put it is in a Django management command.

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


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        c = CopyMapping(
            # Give it the model
```

```
            Person,
            # 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')
        )
        # Then save it.
        c.save()
```

Run your loader and that's it.

```
$ python manage.py mymanagementcommand
Loading CSV to Person
3 records loaded
```

Like I said, that's it!

CHAPTER 4

`CopyMapping` API

class `CopyMapping`(*model*, *csv_path*, *mapping*[, *using=None*, *delimiter=',*, *null=None*, *encoding=None*, *static_mapping=None*])

The following are the arguments and keywords that may be used during instantiation of `CopyMapping` objects.

Argument	Description
`model`	The target model, *not* an instance.
`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.
`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.

CHAPTER 5

`save()` keyword arguments

`CopyMapping.`**`save`**`(`[*silent=False*, *stream=sys.stdout*]`)`

The `save()` method also accepts keywords. These keywords are used for controlling output logging and error handling.

Keyword Arguments	Description
`silent`	By default, non-fatal error notifications are printed to `sys.stdout`, but this keyword may be set to disable these notifications.
`stream`	Status information will be written to this file handle. Defaults to using `sys.stdout`, but any object with a `write` method is supported.

CHAPTER 6

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


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

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 myapp.fields import MyIntegerField

class Person(models.Model):
    name = models.CharField(max_length=500)
    value = MyIntegerField()
```

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

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

    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()`.

CHAPTER 7

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

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

And your loader would look like this:

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


class Command(BaseCommand):

    def handle(self, *args, **kwargs):
        c = CopyMapping(
            # Give it the model
            Person,
            # 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'),
            static_mapping = {
                'source_csv': 'data.csv'
            }
        )
```

```
        # Then save it.
        c.save()
```

CHAPTER 8

Extending with hooks

The `CopyMapping` loader includes optional hooks run before and after the COPY statement that loads your CSV into a temporary table and again before and again the INSERT statement that then slots it into your model.

If you have extra steps or more complicated logic you'd like to work into a loading routine, these 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 as you normally would its parent

```
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 9

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

C

S