Writing Django Extensions for PostgreSQL

Jonathan S. Katz
Exco Ventures
Postgres Open – Sep 15, 2011
Viewer Discretion is Advised

• This talk is targeted for:
  – Django developers
  – Framework developers who want to apply these concepts (e.g. to Rails)
  – DBAs interested in how to get their developers to take advantage of PostgreSQL features
  – Those who know the term “ORM” (object-relational mapper)

• Examples will be using Python and Django. And SQL.
Motivation: My Lament

• I love Postgres
  – ...and I love SQL

• I love web development
  – ...and I like using some ORMs

• I hate when my ORM cannot easily use a Postgres feature or data type
Example: Arrays

- Arrays are fundamental in programming, right?

```sql
SELECT '{1,2,3,5}' AS a; -- SQL
```

```python
a = [1,2,3,4,5]  # Python
```
Example: Arrays

- So why is this such a pain in my ORM?

```python
account = Account.objects.get(pk=1)
account.lotto_numbers = [1, 5, 6, 8, 11]
account.save()  # will fail horrifically
# grrr...
cursor = connection.cursor()
sql = "UPDATE account SET lotto_numbers = '{\%s}' WHERE id = \%s"
cursor.execute(sql, ('{1,5,6,8,11}', account.id,))
```
ActiveRecord has the solution!

• (ActiveRecord = ORM derived from Rails)

    serialize :lotto_numbers, Array

• ...and now, you can reinstantiate all your data as any Ruby class!

• ...oh wait, that just works one way
The Problem is Not The Tool

• Many frameworks support additional components to extend functionality
  – But many of these do not pertain to the database
...the problem is the tool

• However, some frameworks make it difficult to write ORM extensions

• e.g., using “TIMESTAMP WITH TIME ZONE” by default in ActiveRecord

```ruby
ActiveRecord::Base.connection.native_database_types[:datetime] = {
  :name => 'timestamp with time zone'
}
```

• One line, but
  – Zero documentation on how to do this
  – “Hack”
  – Still had some data type conversion issues
Not All Tools Are The Same

• Enter Django
Django is Extensible

• Every core component of Django is designed to be extended
  – “nuance” of Python? :-)
• Writing Django extensions is pretty well documented
  – E.g. model fields: https://docs.djangoproject.com/en/1.3/howto/custom-model-fields/
  – (Still helpful to look at source code)
Enough Talk, We Want Action!

• Does Django support Postgres arrays natively?  
  – No.

• But in five minutes it will...
My Algorithm

1. Understand how data type is represented in PostgreSQL
2. Understand how data type is represented in Python
3. Write Django ⇔ PostgreSQL adapter
4. Write Django form field ⇔ Django model field adapter
Key Methods Inherited from 
models.Field

• `db_type(self, connection)`
  – Defines database data type, based on connection (e.g. Postgres, MySQL, etc.)

• `to_python(self, value)`
  – Mapper from database data type to Python data type
  – Use to put it in most convenient Python type, not display type (e.g. HTML)

• `get_prep_value(self, value)`
  – Python representation => Postgres representation

• `get_prep_db_value(self, value, connection, prepared=False)`
  – `get_prep_value`, but database specific
#1: PostgreSQL Integer Arrays

integer[]

CREATE TABLE (  
id serial,  
lotto_numbers integer[
  ]
);

• Can also limit size of the array, e.g. 6
#2: Python Arrays

• i.e., Python “lists”

```python
a = [1, 2, 3]
b = [4, 'a', True]
```

• We will have to make sure to sanitize our data
from django.db import models
import forms # get forms.IntegerArrayField

class IntegerArrayField(models.Field):
    description = "Use PostgreSQL integer arrays"
    __metaclass__ = models.SubfieldBase

    def __init__(self, *args, **kwargs):
        super(IntegerArrayField, self).__init__(*args, **kwargs)

def db_type(self, connection):
    return 'integer[]'

def formfield(self, **kwargs):
    defaults = {'form_class': forms.IntegerArrayField}
    defaults.update(kwargs)
    return super(IntegerArrayField, self).formfield(**defaults)

def get_prep_value(self, value):
    if isinstance(value, list):
        db_value = str(value)
        db_value = re.sub(r'\[', '{', db_value)
        db_value = re.sub(r'\]', '}', db_value)
        return db_value
    elif isinstance(value, (str, unicode)):
        if not value: return None
        return value

def to_python(self, value):
    if isinstance(value, list):
        return value
    elif isinstance(value, (str, unicode)):
        if not value: return None
        value = re.sub(r'\[\]', '', value).split(',,
        return map(lambda x: int(x), value)
Starting Off: Initial Declarations

class IntegerArrayField(models.Field):
    description = "Use PostgreSQL integer arrays"
    __metaclass__ = models.SubfieldBase

    def __init__(self, *args, **kwargs):
        super(IntegerArrayField, self).__init__(*args, **kwargs)
The Data Type

def db_type(self, connection):
    return 'integer[]'
The Mapping

def get_prep_value(self, value):
    if isinstance(value, list):
        db_value = str(value)
        db_value = re.sub(r'\[', '{', db_value)
        db_value = re.sub(r'\]', '}', db_value)
        return db_value
    elif isinstance(value, (str, unicode)):
        if not value: return None
        return value

def to_python(self, value):
    if isinstance(value, list):
        return value
    elif isinstance(value, (str, unicode)):
        if not value: return None
        value = re.sub(r'\{\}|\}', '', value).split(',,')
        return map(lambda x: int(x), value)
If You Use “south”

• (If you don’t, you should – schema + data migration manager for Django)

• One extra step:

```python
from south.modelsinspector import add_introspection_rules

add_introspection_rules([], ['^main\.models\n  \.IntegerArrayField'])
# where main.models=IntegerArrayField is the module location of
# your custom fields
```
#4: Playing Nicely with Forms

def formfield(self, **kwargs):
    defaults = {'form_class': forms.IntegerArrayField}
    defaults.update(kwargs)
    return super(IntegerArrayField, self).formfield(**defaults)

• Where did we define forms.IntegerArrayField?
forms.IntegerArrayField

class IntegerArrayField(forms.Field):

    def __init__(self, **kwargs):
        super(IntegerArrayField, self).__init__(**kwargs)

    def prepare_value(self, self, value):
        if isinstance(value, list):
            return re.sub(r'\[|\]', '', str(value))
        return value

    def validate(self, self, value):
        super(IntegerArrayField, self).validate(value)
        if not re.search('^\s,0-9]*$', value):
            raise forms.ValidationError, "Please use only integers in your data"
Integer Arrays In Action

• Let’s see what this looks like in a Django app
Nota Bene

• I took a few extra steps when creating this
  – Playing nicely with forms helps users

• I also took a few risks
  – Data validation in the form class, not the model
Time Intervals

• Motivation: Needed to add on X days to a subscription
• Solution: Create a field that uses PostgreSQL time intervals
class DayIntervalField(models.Field):
    SECS_IN_DAY = 86400

    description = "time interval"
    __metaclass__ = models.SubfieldBase

    def __init__(self, *args, **kwargs):
        super(DayIntervalField, self).__init__(*args, **kwargs)

    def db_type(self, connection):
        return 'interval'

    def get_prep_value(self, value):
        try:
            value = int(value)
            return "%d %s" % (value, 'days')
        except:
            if re.match(r"days$", value):
                return value
            elif value:
                return "%s %s" % (value, 'days')
            else:
                return None
Time Intervals In Action
Enumerations

• Enumerations are great for storing:
  – Classifications
  – States (as in state machines)
  – Labels

• PostgreSQL: each enumeration is its own data type

• Django: is it possible to create a generic enumeration field?
class EnumField(models.Field):
    description = "enumerated type"

    def __init__(self, *args, **kwargs):
        self.enum = kwargs['enum']
        del kwargs['enum']
        super(EnumField, self).__init__(*args, **kwargs)

    def db_type(self, self, connection):
        return self.enum

• Base class for enumerations
class MoodEnumField(EnumField):
    description = 'enumerated type for moods'

    def __init__(self, *args, **kwargs):
        self.enum = 'moods'  # key change
        kwargs['enum'] = self.enum
        super(MoodEnumField, self).__init__(*args, **kwargs)
Example

class Profile(models.Model):
    MOODS = (
        ('happy', 'Happy'),
        ('sad', 'Sad'),
        ('angry', 'Angry'),
        ('confused', 'Confused'),
    )

    name = models.CharField(max_length=255)
    moods = MoodEnumField(choices=MOODS)

• All set?
Little bit more work...

- Need to initialize the type

```python
from django.db import connection, transaction

# this only runs on initialization to make sure that the
# proper types are loaded into the DB before we run our initial
# syncdb command
@transaction.atomic
def initialize_custom_types():
    types = {
        # add your custom types here
        'moods': ('happy', 'sad', 'angry', 'confused'),
    }
    cursor = connection.cursor()
    for custom_type, values in types.items():
        cursor.execute(
            'SELECT EXISTS(SELECT typname FROM pg_type WHERE typname=%s);',
            [custom_type])
        result = cursor.fetchone()
        if (not result[0]):
            # note: have to do it this way because otherwise the ORM string escapes the value, which we
do not want
            # but yes, understand the risks how this is open to a SQL injection attack
            sql = "CREATE TYPE {} AS ENUM %s;".format(custom_type)
            cursor.execute(sql, [values])
    transaction.commit_unless_managed()
```
More Efficient Way?

- Do the benefits outweigh the hassle of enumerated types?
  - Yes for large data sets – storage space + performance
- Can probably find a more efficient way of representing them
Other Types I’ve Completed

• Money
  – Not native to Postgres but very useful

• Point
  – Great for 9.1
  – Issue with queries called with “DISTINCT” due to lack of “=” defined
What I Did Not Cover

• Encapsulating Functionality
  – Fulltext search
  – Functions
  – Extensions

• PostGIS & GeoDjango
  – Many PostGIS specific data type extensions
Conclusion

• It can be hard to have the best of both worlds
  – But it’s worth it!
• Django provides a robust framework for extending itself
  – Which allows app developers to fully utilize Postgres
Code examples: [https://github.com/jkatz/django_postgres_extensions](https://github.com/jkatz/django_postgres_extensions)

- Let’s expand the supported data types

This talk: [http://excoventures.com/talks/django-extensions.pdf](http://excoventures.com/talks/django-extensions.pdf)

Django docs: [https://docs.djangoproject.com/en/1.3/howto/custom-model-fields](https://docs.djangoproject.com/en/1.3/howto/custom-model-fields)

Contact

• jonathan.katz@excoventures.com
• @jkatz05