Writing Django Extensions for PostgreSQL

Jonathan S. Katz
Exco Ventures
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 from Rails)

```ruby
serialize :lotto_numbers, Array
```

- ...and now, you can reinstantiate all your data as any Ruby class? Maybe?
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

• Some ORMs are difficult to extend
  – e.g. “TIMESTAMP WITH TIME ZONE” by default in ActiveRecord
    ```ruby
    ActiveRecord::Base.connection.native_database_types[:datetime] = {
      :name => 'timestamp with time zone'
    }
    ```
• Simple, but
  – Zero documentation on how to do this
  – “Hack”
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 well-documented
  – E.g. model fields: https://docs.djangoproject.com/en/1.3/howto/custom-model-fields/
  – Often helpful to look at source code
Enough Talk, We Want Action!

• Does Django support Postgres arrays natively?

• Perhaps in five minutes...
My Algorithm

1. Determine Postgres data type representation
2. Determine Python data type representation
3. Write Django ↔ PostgreSQL adapter
4. Write Django form field ↔ Django model field adapter
Key Methods from models.Field

- **db_type(self, connection)**
  - Defines database data type, based on connection (e.g. Postgres, MySQL, etc.)

- **to_python(self, value)**
  - maps DB type to Python type
  - Use to for most convenient Python type, not display type (e.g. HTML)
Key Methods from models.Field

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

• `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 define limit to array size
#2: Python Arrays

- i.e., Python “lists”

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

- N.B: need to sanitize data between Postgres and Python
from django.db import models
import forms # get forms.IntegerField

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)
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:

    from south.modelsinspector import add_introspection_rules

    add_introspection_rules([], ['^main\.models \.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, value):
        if isinstance(value, list):
            return re.sub(r'\[|\]', '', str(value))
        return value
    
def validate(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

• [Quick app demo]
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
Hstore – Key-Value Pairs

• Prerequisites: hstore contrib package installed
  – 9.1: CREATE EXTENSION hstore;
  – 8.3-9.0: psql dbname < $INSTALL_PATH/share/contrib/hstore.sql

• Useful for storing amorphous key-value pairs
class HstoreField(models.Field):
    description = "Use PostgreSQL hstore"
    __metaclass__ = models.SubfieldBase

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

    def get_prep_value(self, value):
        if isinstance(value, (str, unicode)):
            pass
        elif isinstance(value, dict):
            values = []
            for key in value.keys():
                values.append('%s="%s" % (key, value[key])

            return ", ".join(values)
Hstore Cont’d

def to_python(self, value):
    if isinstance(value, dict):
        return value
    elif value is None:
        return None
    else:
        value = re.split('"\s*,\s*"', value)
        return dict(map(self._hstore_clean, value))

def _hstore_clean(self, value):
    k, v = value.strip().split('=>')
    k = re.sub('^"|"$', '', k)
    v = re.sub('^"|"$', '', v)
    return [k,v]
Enumerations

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

• PostgreSQL: each enumeration is its own 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
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)
class Profile(models.Model):
    MOODS = (
        ('happy', 'Happy'),
        ('sad', 'Sad'),
        ('angry', 'Angry'),
        ('confused', 'Confused'),
    )

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

• But...
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.autocommit()
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 " + custom_type + " AS ENUM %s;"
            cursor.execute(sql, [values])
            transaction.commit_unless_managed()
```
More Efficient Way?

• Do the benefits outweigh the hassle of enumerated types?
  – Large data sets – good for storage space + performance

• Is there a more efficient way?
  – Open source :-)


Caveat Emptor

• Model filters may not give you expected behavior

• Need to anticipate what data types are presented
  – e.g. str + unicode

• Need to know when data types can change
  – {‘a’: 1 } => {‘a’: ‘1’}
  – Developer’s onus
Demos (Time Permitting)
Other Types I’ve Completed

• Money
  – Postgres agnostic: breaks down monetary type into integer

• 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!

• Apply concepts to other projects – get the most out of your Postgres!
References

• Code examples: (https://github.com/jkatz/django_postgres_extensions)
  – Let’s expand the supported data types
• Django docs: https://docs.djangoproject.com/en/1.3/howto/custom-model-fields
• PostGIS Extensions: GeoDjango: https://docs.djangoproject.com/en/1.3/ref/contrib/gis/
Contact

• jonathan.katz@excoventures.com
• @jkatz05
• Feedback Please!
  – http://2011.pgconf.eu/feedback