Runtime Dynamic Models Documentation

Release 1.0

Will Hardy

October 05, 2016

Contents

The flexibility of Python and Django allow developers to dynamically create models to store and access data using Django's ORM. But you need to be careful if you go down this road, especially if your models are set to change at runtime. This documentation will cover a number of things to consider when making use of runtime dynamic models.

An example implementation of dynamic models is also provided for reference. It is hosted here on GitHub: willhardy/dynamic-models.

This topic was presented at DjangoCon and these notes have been written as supplementary documentation for that talk. The talk can be viewed online here.

1 Defining a dynamic model factory

The basic principle that allows us to create dynamic classes is the built-in function `type()`. Instead of the normal syntax to define a class in Python:

```
class Person(object):
    name = "Julia"
```

The `type()` function can be used to create the same class, here is how the class above looks using the `type()` built-in:

```
Person = type("Person", (object,), {'name': "Julia"})
```

Using `type()` means you can programatically determine the number and names of the attributes that make up the class.

1.1 Django models

Django models can be essentially defined in the same manner, with the one additional requirement that you need to define an attribute called `__module__`. Here is a simple Django model:

```
class Animal(models.Model):
    name = models.CharField(max_length=32)
```

And here is the equivalent class built using `type()`:

```
attrs = {
    'name': models.CharField(max_length=32),
    '__module__': 'myapp.models'
}
Animal = type("Animal", (models.Model,), attrs)
```

Any Django model that can be defined in the normal fashion can be made using `type()`.

1.2 Django's model cache

Django automatically caches model classes when you subclass `models.Model`. If you are generating a model that has a name that may already exist, you should firstly remove the existing cached class.

There is no official, documented way to do this, but current versions of Django allow you to delete the cache entry directly:

```
from django.db.models.loading import cache
try:
    del cache.app_models[appname][modelname]
except KeyError:
    pass
```

Note: When using Django in non-official or undocumented ways, it's highly advisable to write unit tests to ensure that the code does what you indend it to do. This is especially useful when upgrading Django in the future, to ensure that all uses of undocumented features still work with the new version of Django.

1.3 Using the model API

Because the names of model fields may no longer be known to the developer, it makes using Django's model API a little more difficult. There are at least three simple approaches to this problem.

Firstly, you can use Python's `**` syntax to pass a mapping object as a set of keyword arguments. This is not as elegant as the normal syntax, but does the job:

```
kwargs = {'name': "Jenny", 'color': "Blue"}
print People.objects.filter(**kwargs)
```

A second approach is to subclass `django.db.models.query.QuerySet` and provide your own customisations to keep things clean. You can attach the customised `QuerySet` class by overloading the `get_query_set` method of your model manager. Beware however of making things too nonstandard, forcing other developers to learn your new API.

```
from django.db.models.query import QuerySet
from django.db import models

class MyQuerySet(QuerySet):
    def filter(self, *args, **kwargs):
        kwargs.update((args[i],args[i+1]) for i in range(0, len(args), 2))
        return super(MyQuerySet, self).filter(**kwargs)

class MyManager(models.Manager):
    def get_query_set(self):
        return MyQuerySet(self.model)

# XXX Add the manager to your dynamic model...

# Warning: This project uses a customised filter method!
print People.objects.filter(name="Jenny").filter('color', 'blue')
```

A third approach is to simply provide a helper function that creates either a preprepared `kwargs` mapping or returns a `django.db.models.Q` object, which can be fed directly to a queryset as seen above. This would be like creating a new API, but is a little more explicit than subclassing `QuerySet`.

```
from django.db.models import Q

def my_query(*args, **kwargs):
    """ turns my_query(key, val, key, val, key=val) into a Q object. """
    kwargs.update((args[i],args[i+1]) for i in range(0, len(args), 2))
    return Q(**kwargs)

print People.objects.filter(my_query('color', 'blue', name="Jenny"))
```

1.4 What comes next?

Although this is enough to define a Django model class, if the model isn't in existence when `syncdb` is run, no respective database tables will be created. The creation and migration of database tables is covered in database migration.

Also relevant is the appropriately time regeneration of the model class, (*especially* if you want to host using more than one server) see model migration and, if you would like to edit the dynamic models in Django's admin, admin migration.

2 Model migration

Model migration is simplest when you just regenerate the model using the usual factory function. The problem is when the change has been instigated in another process, the current process will not know that the model class needs to be regenerated again. Because it would generally be insane to regenerate the model on every view, you will need to send a message to other processes, to inform them that their cached model class is no longer valid.

The most efficient way to check equality is to make a hash describing the dynamic model, and let processes compare their hash with the latest version.

2.1 Generating a hash

You're free to do it however you like, just make sure it is deterministic, ie you explicitly order the encoded fields.

In the provided example, a json representation of the fields relevant to dynamic model is used to create a hash. For example:

```
from django.utils import simplejson
from django.utils.hashcompat import md5_constructor

i = my_instance_that_defines_a_dynamic_model
val = (i.slug, i.name, [i.fields for i in self.field_set.all())
print md5_constructor(simplejson.dumps(val)).hexdigest()
```

If you use a dict-like object, make sure you set `sort_keys=True` when calling `json.dumps`.

2.2 Synchronising processes

The simplest way to ensure a valid model class is provided to a view is to validate the hash every time it is accessed. This means, each time a view would like to use the dynamic model class, the factory function checks the hash against one stored in a shared data store. Whenever a model class is generated, the shared store's hash is updated.

Generally you will use something fast for this, for example memcached or redis. As long as all processes have access to the same data store, this approach will work.

Of course, there can be race conditions. If generating the dynamic model class takes longer in one process then its hash may overwrite that from a more recent version.

In that case, the only prevention may be to either using the database as a shared store, keeping all related changes to the one transaction, or by manually implementing a locking strategy.

Dynamic models are surprisingly stable when the definitions change rarely. But I cannot vouch for their robustness where migrations are often occuring, in more than one process or thread.

3 Database schema migration

As mentioned earlier, creating model classes at runtime means that the relevant database tables will not be created by Django when running syncdb; you will have to create them yourself. Additionally, if your dynamic models are likely to change you are going to have to handle database schema (and data) migration.

3.1 Schema and data migrations with South

Thankfully, South has a reliable set of functions to handle schema and database migrations for Django projects. When used in development, South can suggest migrations but does not attempt to automatically apply them without interaction from the developer. This can be different for your system, if you are able to recognised the required migration actions with 100% confidence, there should be no issue with automatically running schema and data migrations. That said, any automatic action is a dangerous one, be sure to err on the side of caution and avoid destructive operations as much as possible.

Here is a (perfectly safe) way to create a table from your dynamic model.

```
from south.db import db

model_class = generate_my_model_class()
fields = [(f.name, f) for f in model_class._meta.local_fields]
table_name = model_class._meta.db_table

db.create_table(table_name, fields)

# some fields (eg GeoDjango) require additional SQL to be executed
db.execute_deferred_sql()
```

Basic schema migration can also be easily performed. Note that if the column type changes in a way that requires data conversion, you may have to migrate the data manually. Remember to run `execute_deferred_sql` after adding a new table or column, to handle a number of special model fields (eg `ForeignKey`, `ManyToManyField`, GeoDjango fields etc).

```
db.add_column(table_name, name, field)
db.execute_deferred_sql()

db.rename_column(table_name, old, new)
db.rename_table(old_table_name, new_table_name)

db.alter_column(table_name, name, field)
```

Indexes and unique constraints may need to be handled separately:

```
db.create_unique(table_name, columns)
db.delete_unique(table_name, columns)
db.create_index(table_name, column_names, unique=False)
db.delete_index(table_name, column_name)

db.create_primary_key(table_name, columns) # err... does your schema
db.delete_primary_key(table_name)          # really need to be so dynamic?
```

If you really need to delete tables and columns, you can do that too. It's a good idea to avoid destructive operations until they're necessary. Leaving orphaned tables and columns for a period of time and cleaning them at a later date is perfectly acceptable. You may want to have your own deletion policy and process, depending on your needs.

```
db.delete_table(table_name)
db.delete_column(table_name, field)
```

Note: Note that this South functionality is in the process of being merged into Django core. It will hopefully land in trunk in the near future.

3.2 Timing the changes

Using Django's standard signals, you can perform the relevant actions to migrate the database schema at the right time. For example, create the new table on `post_save` when `created=True`.

You may also wish to run some conditional migrations at startup. For that you'll need to use the `class_prepared` signal, but wait until the models that your factory function require have all been prepared. The following function handles this timing. Place it in your `models.py` before any of the required models have been defined and it will call the given function when the time is right:

```
when_classes_prepared(app_label, req_models, builder_fn)
```

The function's implementation can be found in the example code, in `surveymaker.utils`.

Another useful feature is to be able to identify when a column rename is required. If your dynamic models are defined by Django models, it may be as simple as determining if an attribute on a model instance has been changed. You can do this with a combination of `pre_save` and `post_save` signals (see `surveymaker.signals` in example code for an example of this) or you can override the *__init__* method of the relevant model to store the original values when an instance is created. The `post_save` signal can then detect if a change was made and trigger the column rename.

If you're concerned about failed migrations causing an inconsistent system state you may want to ensure that the migrations are in the same transaction as the changes that cause them.

3.3 Introspection

It may be useful to perform introspection, especially if you leave "deleted" tables and columns lying around, or if naming conflicts are possible (but please try to make them impossible). This means, the system will react in the way you want it to, for example by renaming or deleting the existing tables or by aborting the proposed schema migration.

Django provides an interface for its supported databases, where existing table names and descriptions can be easily discovered:

```python
from django.db.connection import introspection
from django.db import connection

name = introspection.table_name_converter(table_name)

# Is my table already there?
print name in introspection.table_names()

description = introspection.get_table_description(connection.cursor(), name)
db_column_names = [row[0] for row in description]

# Is my field's column already there?
print myfield.column in db_column_names
```

Note that this is limited to standard field types, some fields aren't exactly columns.

4 Admin migration

When using Django's admin, you will need to update the admin site when your dynamic model changes. This is one area where it is not entirely straightforward, because we are now dealing with a "third-party app" that does not expect the given models to change. That said, there are ways around almost all of the main problems.

4.1 Unregistering

Calling `site.unregister` will not suffice, the model will have changed and the admin will not find the old dynamic model when given the new one. The solution is however straightforward, we search manually using the name of the old dynamic model, which will often be the same as the new model's name.

The following code ensures the model is unregistered:

```
from django.contrib import admin
site = admin.site
model = my_dynamic_model_factory()

for reg_model in site._registry.keys():
    if model._meta.db_table == reg_model._meta.db_table:
        del site._registry[reg_model]

# Try the regular approach too, but this may be overdoing it
try:
    site.unregister(model)
except NotRegistered:
    pass
```

Note: Again, this is accessing undocumented parts of Django's core. Please write a unit test to confirm that it works, so that you will notice any backwards incompatible changes in a future Django update.

4.2 Clearing the URL cache

Even though you may have successfully re-registered a newly changed model, Django caches URLs as soon as they are first loaded. The following idiomatic code will reset the URL cache, allowing new URLs to take effect.

```
from django.conf import settings
from django.utils.importlib import import_module
from django.core.urlresolvers import clear_url_caches

reload(import_module(settings.ROOT_URLCONF))
clear_url_caches()
```

4.3 Timing Admin updates

Once again, Django's signals can be used to trigger an update of the Admin. Unfortunately, this cannot be done when the change takes place in another process.

Because Django's admin doesn't use your factory function to access the model class (it uses the cached version), it cannot check the hash for validity nor can it rebuild when necessary.

This isn't a problem if your Admin site is carefully place on a single server and all meaningful changes take place on the same instance. In reality, it's not always on a single server and background processes and tasks need to be able to alter the schema.

A fix may involve pushing the update to the admin process, be it a rudimentary hook URL, or something much more sophisticated.