django-polymorphic Documentation

Release 0.9.2

Bert Constantin, Chris Glass, Diederik van der Boor

July 08, 2016

Contents

Django-polymorphic simplifies using inherited models in Django projects. When a query is made at the base model, the inherited model classes are returned.

When we store models that inherit from a `Project` model...

```
>>> Project.objects.create(topic="Department Party")
>>> ArtProject.objects.create(topic="Painting with Tim", artist="T. Turner")
>>> ResearchProject.objects.create(topic="Swallow Aerodynamics", supervisor="Dr. Winter")
```

...and want to retrieve all our projects, the subclassed models are returned!

```
>>> Project.objects.all()
    [ <Project:         id 1, topic "Department Party">,
      <ArtProject:      id 2, topic "Painting with Tim", artist "T. Turner">,
      <ResearchProject: id 3, topic "Swallow Aerodynamics", supervisor "Dr. Winter"> ]
```

Using vanilla Django, we get the base class objects, which is rarely what we wanted:

```
>>> Project.objects.all()
    [ <Project: id 1, topic "Department Party">,
      <Project: id 2, topic "Painting with Tim">,
      <Project: id 3, topic "Swallow Aerodynamics"> ]
```

CHAPTER 1

Features

- Full admin integration.
- ORM integration:
- Support for ForeignKey, ManyToManyField, OneToOneField descriptors.
- Support for proxy models.
- Filtering/ordering of inherited models (`ArtProject___artist`).
- Filtering model types: `instance_of(...)` and `not_instance_of(...)`
- Combining querysets of different models (`qs3 = qs1 | qs2`)
- Support for custom user-defined managers.
- Uses the minimum amount of queries needed to fetch the inherited models.
- Disabling polymorphic behavior when needed.

CHAPTER 2

Getting started

2.1 Quickstart

Install the project using:

```
pip install django-polymorphic
```

Update the settings file:

```
INSTALLED_APPS += (
    'polymorphic',
    'django.contrib.contenttypes',
)
```

The current release of *django-polymorphic* supports Django 1.4 till 1.9 and Python 3 is supported.

2.1.1 Making Your Models Polymorphic

Use `PolymorphicModel` instead of Django's `models.Model`, like so:

```
from polymorphic.models import PolymorphicModel

class Project(PolymorphicModel):
    topic = models.CharField(max_length=30)

class ArtProject(Project):
    artist = models.CharField(max_length=30)

class ResearchProject(Project):
    supervisor = models.CharField(max_length=30)
```

All models inheriting from your polymorphic models will be polymorphic as well.

2.1.2 Using Polymorphic Models

Create some objects:

```
>>> Project.objects.create(topic="Department Party")
>>> ArtProject.objects.create(topic="Painting with Tim", artist="T. Turner")
>>> ResearchProject.objects.create(topic="Swallow Aerodynamics", supervisor="Dr. Winter")
```

Get polymorphic query results:

```
>>> Project.objects.all()
[ <Project:          id 1, topic "Department Party">,
  <ArtProject:       id 2, topic "Painting with Tim", artist "T. Turner">,
  <ResearchProject: id 3, topic "Swallow Aerodynamics", supervisor "Dr. Winter"> ]
```

Use `instance_of` or `not_instance_of` for narrowing the result to specific subtypes:

```
>>> Project.objects.instance_of(ArtProject)
[ <ArtProject:       id 2, topic "Painting with Tim", artist "T. Turner"> ]
```

```
>>> Project.objects.instance_of(ArtProject) | Project.objects.instance_of(ResearchProject)
[ <ArtProject:       id 2, topic "Painting with Tim", artist "T. Turner">,
  <ResearchProject: id 3, topic "Swallow Aerodynamics", supervisor "Dr. Winter"> ]
```

Polymorphic filtering: Get all projects where Mr. Turner is involved as an artist or supervisor (note the three underscores):

```
>>> Project.objects.filter(Q(ArtProject___artist='T. Turner') | Q(ResearchProject___supervisor='T. Tu
[ <ArtProject:       id 2, topic "Painting with Tim", artist "T. Turner">,
  <ResearchProject: id 4, topic "Color Use in Late Cubism", supervisor "T. Turner"> ]
```

This is basically all you need to know, as *django-polymorphic* mostly works fully automatic and just delivers the expected results.

Note: When using the `dumpdata` management command on polymorphic tables (or any table that has a reference to `ContentType`), include the `--natural` flag in the arguments. This makes sure the `ContentType` models will be referenced by name instead of their primary key as that changes between Django instances.

Note: While *django-polymorphic* makes subclassed models easy to use in Django, we still encourage to use them with caution. Each subclassed model will require Django to perform an `INNER JOIN` to fetch the model fields from the database. While taking this in mind, there are valid reasons for using subclassed models. That's what this library is designed for!

2.2 Django admin integration

Off course, it's possible to register individual polymorphic models in the Django admin interface. However, to use these models in a single cohesive interface, some extra base classes are available.

The polymorphic admin interface works in a simple way:

- The add screen gains an additional step where the desired child model is selected.
- The edit screen displays the admin interface of the child model.
- The list screen still displays all objects of the base class.

The polymorphic admin is implemented via a parent admin that forwards the *edit* and *delete* views to the `ModelAdmin` of the derived child model. The *list* page is still implemented by the parent model admin.

Both the parent model and child model need to have a `ModelAdmin` class. Only the `ModelAdmin` class of the parent/base model has to be registered in the Django admin site.

2.2.1 The parent model

The parent model needs to inherit `PolymorphicParentModelAdmin`, and implement the following:

- `base_model` should be set
- `child_models` or `get_child_models()` should return a list with (Model, ModelAdmin) tuple.

The exact implementation can depend on the way your module is structured. For simple inheritance situations, `child_models` is the best solution. For large applications, `get_child_models()` can be used to query a plugin registration system.

By default, the non_polymorphic() method will be called on the queryset, so only the Parent model will be provided to the list template. This is to avoid the performance hit of retrieving child models.

This can be controlled by setting the `polymorphic_list` property on the parent admin. Setting it to True will provide child models to the list template.

Note: If you are using non-integer primary keys in your model, you have to edit `pk_regex`, for example `pk_regex = '([\w-]+)'` if you use UUIDs. Otherwise you cannot change model entries.

2.2.2 The child models

The admin interface of the derived models should inherit from `PolymorphicChildModelAdmin`. Again, `base_model` should be set in this class as well. This class implements the following features:

- It corrects the breadcrumbs in the admin pages.
- It extends the template lookup paths, to look for both the parent model and child model in the `admin/app/model/change_form.html` path.
- It allows to set `base_form` so the derived class will automatically include other fields in the form.
- It allows to set `base_fieldsets` so the derived class will automatically display any extra fields.

The standard `ModelAdmin` attributes `form` and `fieldsets` should rather be avoided at the base class, because it will hide any additional fields which are defined in the derived model. Instead, use the `base_form` and `base_fieldsets` instead. The `PolymorphicChildModelAdmin` will automatically detect the additional fields that the child model has, display those in a separate fieldset.

2.2.3 Polymorphic Inlines

To add a polymorphic child model as an Inline for another model, add a field to the inline's readonly_fields list formed by the lowercased name of the polymorphic parent model with the string "_ptr" appended to it. Otherwise, trying to save that model in the admin will raise an AttributeError with the message "can't set attribute".

2.2.4 Example

The models are taken from *Advanced features*.

```
from django.contrib import admin
from polymorphic.admin import PolymorphicParentModelAdmin, PolymorphicChildModelAdmin
from .models import ModelA, ModelB, ModelC, StandardModel


class ModelAChildAdmin(PolymorphicChildModelAdmin):
    """ Base admin class for all child models """
    base_model = ModelA
```

```
    # By using these `base_...` attributes instead of the regular ModelAdmin `form` and `fieldsets`,
    # the additional fields of the child models are automatically added to the admin form.
    base_form = ...
    base_fieldsets = (
        ...
    )


class ModelBAdmin(ModelAChildAdmin):
    base_model = ModelB
    # define custom features here


class ModelCAdmin(ModelBAdmin):
    base_model = ModelC
    # define custom features here


class ModelAParentAdmin(PolymorphicParentModelAdmin):
    """ The parent model admin """
    base_model = ModelA
    child_models = (
        (ModelB, ModelBAdmin),
        (ModelC, ModelCAdmin),
    )


class ModelBInline(admin.StackedInline):
    model = ModelB
    fk_name = 'modelb'
    readonly_fields = ['modela_ptr']


class StandardModelAdmin(admin.ModelAdmin):
    inlines = [ModelBInline]


# Only the parent needs to be registered:
admin.site.register(ModelA, ModelAParentAdmin)
admin.site.register(StandardModel, StandardModelAdmin)
```

2.3 Performance Considerations

Usually, when Django users create their own polymorphic ad-hoc solution without a tool like *django-polymorphic*, this usually results in a variation of

```
result_objects = [ o.get_real_instance() for o in BaseModel.objects.filter(...) ]
```

which has very bad performance, as it introduces one additional SQL query for every object in the result which is not of class `BaseModel`. Compared to these solutions, *django-polymorphic* has the advantage that it only needs 1 SQL query *per object type*, and not *per object*.

The current implementation does not use any custom SQL or Django DB layer internals - it is purely based on the standard Django ORM. Specifically, the query:

```
result_objects = list( ModelA.objects.filter(...) )
```

performs one SQL query to retrieve `ModelA` objects and one additional query for each unique derived class occurring in result_objects. The best case for retrieving 100 objects is 1 SQL query if all are class `ModelA`. If 50 objects are `ModelA` and 50 are `ModelB`, then two queries are executed. The pathological worst case is 101 db queries if result_objects contains 100 different object types (with all of them subclasses of `ModelA`).

2.3.1 ContentType retrieval

When fetching the `ContentType` class, it's tempting to read the `object.polymorphic_ctype` field directly. However, this performs an additional query via the `ForeignKey` object to fetch the `ContentType`. Instead, use:

```
from django.contrib.contenttypes.models import ContentType

ctype = ContentType.objects.get_for_id(object.polymorphic_ctype_id)
```

This uses the `get_for_id()` function which caches the results internally.

2.3.2 Database notes

Current relational DBM systems seem to have general problems with the SQL queries produced by object relational mappers like the Django ORM, if these use multi-table inheritance like Django's ORM does. The "inner joins" in these queries can perform very badly. This is independent of django_polymorphic and affects all uses of multi table Model inheritance.

Please also see this post (and comments) from Jacob Kaplan-Moss.

CHAPTER 3

Advanced topics

3.1 Migrating existing models to polymorphic

Existing models can be migrated to become polymorphic models. During the migrating, the `polymorphic_ctype` field needs to be filled in.

This can be done in the following steps:

1. Inherit your model from `PolymorphicModel`.
2. Create a Django migration file to create the `polymorphic_ctype_id` database column.
3. Make sure the proper `ContentType` value is filled in.

3.1.1 Filling the content type value

The following Python code can be used to fill the value of a model:

```
from django.contrib.contenttypes.models import ContentType
from myapp.models import MyModel

new_ct = ContentType.objects.get_for_model(MyModel)
MyModel.objects.filter(polymorphic_ctype__isnull=True).update(polymorphic_ctype=new_ct)
```

The creation and update of the `polymorphic_ctype_id` column can be included in a single Django migration. For example:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.db import migrations, models


def forwards_func(apps, schema_editor):
    MyModel = apps.get_model('myapp', 'MyModel')
    ContentType = apps.get_model('contenttypes', 'ContentType')

    new_ct = ContentType.objects.get_for_model(MyModel)
    MyModel.objects.filter(polymorphic_ctype__isnull=True).update(polymorphic_ctype=new_ct)


def backwards_func(apps, schema_editor):
    pass
```

```
class Migration(migrations.Migration):

    dependencies = [
        ('contenttypes', '0001_initial'),
        ('myapp', '0001_initial'),
    ]

    operations = [
        migrations.AddField(
            model_name='mymodel',
            name='polymorphic_ctype',
            field=models.ForeignKey(related_name='polymorphic_myapp.mymodel_set+', editable=False, to
        ),
        migrations.RunPython(forwards_func, backwards_func),
    ]
```

It's recommended to let `makemigrations` create the migration file, and include the `RunPython` manually before running the migration.

3.2 Advanced features

In the examples below, these models are being used:

```
from django.db import models
from polymorphic.models import PolymorphicModel

class ModelA(PolymorphicModel):
    field1 = models.CharField(max_length=10)

class ModelB(ModelA):
    field2 = models.CharField(max_length=10)

class ModelC(ModelB):
    field3 = models.CharField(max_length=10)
```

3.2.1 Filtering for classes (equivalent to python's isinstance()):

```
>>> ModelA.objects.instance_of(ModelB)
.
[ <ModelB: id 2, field1 (CharField), field2 (CharField)>,
  <ModelC: id 3, field1 (CharField), field2 (CharField), field3 (CharField)> ]
```

In general, including or excluding parts of the inheritance tree:

```
ModelA.objects.instance_of(ModelB [, ModelC ...])
ModelA.objects.not_instance_of(ModelB [, ModelC ...])
```

You can also use this feature in Q-objects (with the same result as above):

```
>>> ModelA.objects.filter( Q(instance_of=ModelB) )
```

3.2.2 Polymorphic filtering (for fields in inherited classes)

For example, cherrypicking objects from multiple derived classes anywhere in the inheritance tree, using Q objects (with the syntax: `exact model name + three _ + field name`):

```
>>> ModelA.objects.filter(  Q(ModelB___field2 = 'B2') | Q(ModelC___field3 = 'C3')  )
.
[ <ModelB: id 2, field1 (CharField), field2 (CharField)>,
  <ModelC: id 3, field1 (CharField), field2 (CharField), field3 (CharField)> ]
```

3.2.3 Combining Querysets

Querysets could now be regarded as object containers that allow the aggregation of different object types, very similar to python lists - as long as the objects are accessed through the manager of a common base class:

```
>>> Base.objects.instance_of(ModelX) | Base.objects.instance_of(ModelY)
.
[ <ModelX: id 1, field_x (CharField)>,
  <ModelY: id 2, field_y (CharField)> ]
```

3.2.4 ManyToManyField, ForeignKey, OneToOneField

Relationship fields referring to polymorphic models work as expected: like polymorphic querysets they now always return the referred objects with the same type/class these were created and saved as.

E.g., if in your model you define:

```
field1 = OneToOneField(ModelA)
```

then field1 may now also refer to objects of type `ModelB` or `ModelC`.

A ManyToManyField example:

```
# The model holding the relation may be any kind of model, polymorphic or not
class RelatingModel(models.Model):
    many2many = models.ManyToManyField('ModelA')  # ManyToMany relation to a polymorphic model

>>> o=RelatingModel.objects.create()
>>> o.many2many.add(ModelA.objects.get(id=1))
>>> o.many2many.add(ModelB.objects.get(id=2))
>>> o.many2many.add(ModelC.objects.get(id=3))

>>> o.many2many.all()
[ <ModelA: id 1, field1 (CharField)>,
  <ModelB: id 2, field1 (CharField), field2 (CharField)>,
  <ModelC: id 3, field1 (CharField), field2 (CharField), field3 (CharField)> ]
```

3.2.5 Using Third Party Models (without modifying them)

Third party models can be used as polymorphic models without restrictions by subclassing them. E.g. using a third party model as the root of a polymorphic inheritance tree:

```
from thirdparty import ThirdPartyModel
```

```
class MyThirdPartyBaseModel(PolymorphicModel, ThirdPartyModel):
    pass    # or add fields
```

Or instead integrating the third party model anywhere into an existing polymorphic inheritance tree:

```
class MyBaseModel(SomePolymorphicModel):
    my_field = models.CharField(max_length=10)

class MyModelWithThirdParty(MyBaseModel, ThirdPartyModel):
    pass    # or add fields
```

3.2.6 Non-Polymorphic Queries

If you insert `.non_polymorphic()` anywhere into the query chain, then django_polymorphic will simply leave out the final step of retrieving the real objects, and the manager/queryset will return objects of the type of the base class you used for the query, like vanilla Django would (`ModelA` in this example).

```
>>> qs=ModelA.objects.non_polymorphic().all()
>>> qs
[ <ModelA: id 1, field1 (CharField)>,
  <ModelA: id 2, field1 (CharField)>,
  <ModelA: id 3, field1 (CharField)> ]
```

There are no other changes in the behaviour of the queryset. For example, enhancements for `filter()` or `instance_of()` etc. still work as expected. If you do the final step yourself, you get the usual polymorphic result:

```
>>> ModelA.objects.get_real_instances(qs)
[ <ModelA: id 1, field1 (CharField)>,
  <ModelB: id 2, field1 (CharField), field2 (CharField)>,
  <ModelC: id 3, field1 (CharField), field2 (CharField), field3 (CharField)> ]
```

3.2.7 About Queryset Methods

- `annotate()` and `aggregate()` work just as usual, with the addition that the `ModelX___field` syntax can be used for the keyword arguments (but not for the non-keyword arguments).
- `order_by()` similarly supports the `ModelX___field` syntax for specifying ordering through a field in a submodel.
- `distinct()` works as expected. It only regards the fields of the base class, but this should never make a difference.
- `select_related()` works just as usual, but it can not (yet) be used to select relations in inherited models (like `ModelA.objects.select_related('ModelC___fieldxy')`)
- `extra()` works as expected (it returns polymorphic results) but currently has one restriction: The resulting objects are required to have a unique primary key within the result set - otherwise an error is thrown (this case could be made to work, however it may be mostly unneeded).. The keyword-argument "polymorphic" is no longer supported. You can get back the old non-polymorphic behaviour by using `ModelA.objects.non_polymorphic().extra(...)`.
- `get_real_instances()` allows you to turn a queryset or list of base model objects efficiently into the real objects. For example, you could do `base_objects_queryset=ModelA.extra(...).non_polymorphic()` and then call `real_objects=base_objects_queryset.get_real_instances()`. Or alternatively .''real_objects=ModelA.objects.get_real_instances(base_objects_queryset_or_object_list)''

- `values()` & `values_list()` currently do not return polymorphic results. This may change in the future however. If you want to use these methods now, it's best if you use `Model.base_objects.values...` as this is guaranteed to not change.
- `defer()` and `only()` work as expected. On Django 1.5+ they support the `ModelX___field` syntax, but on Django 1.4 it is only possible to pass fields on the base model into these methods.

3.2.8 Using enhanced Q-objects in any Places

The queryset enhancements (e.g. `instance_of`) only work as arguments to the member functions of a polymorphic queryset. Occasionally it may be useful to be able to use Q objects with these enhancements in other places. As Django doesn't understand these enhanced Q objects, you need to transform them manually into normal Q objects before you can feed them to a Django queryset or function:

```
normal_q_object = ModelA.translate_polymorphic_Q_object( Q(instance_of=Model2B) )
```

This function cannot be used at model creation time however (in models.py), as it may need to access the ContentTypes database table.

3.2.9 Nicely Displaying Polymorphic Querysets

In order to get the output as seen in all examples here, you need to use the `ShowFieldType` class mixin:

```
from polymorphic.showfields import PolymorphicModel, ShowFieldType

class ModelA(ShowFieldType, PolymorphicModel):
    field1 = models.CharField(max_length=10)
```

You may also use `ShowFieldContent` or `ShowFieldTypeAndContent` to display additional information when printing querysets (or converting them to text).

When showing field contents, they will be truncated to 20 characters. You can modify this behaviour by setting a class variable in your model like this:

```
class ModelA(ShowFieldType, PolymorphicModel):
    polymorphic_showfield_max_field_width = 20
    ...
```

Similarly, pre-V1.0 output formatting can be re-estated by using `polymorphic_showfield_old_format = True`.

3.2.10 Restrictions & Caveats

- Database Performance regarding concrete Model inheritance in general. Please see the *Performance Considerations*.
- Queryset methods `values()`, `values_list()`, and `select_related()` are not yet fully supported (see above). `extra()` has one restriction: the resulting objects are required to have a unique primary key within the result set.
- Diamond shaped inheritance: There seems to be a general problem with diamond shaped multiple model inheritance with Django models (tested with V1.1 - V1.3). An example is here: http://code.djangoproject.com/ticket/10808. This problem is aggravated when trying to enhance models.Model by subclassing it instead of modifying Django core (as we do here with PolymorphicModel).

- The enhanced filter-definitions/Q-objects only work as arguments for the methods of the polymorphic querysets. Please see above for `translate_polymorphic_Q_object`.
- When using the `dumpdata` management command on polymorphic tables (or any table that has a reference to `ContentType`), include the `--natural` flag in the arguments.

3.3 Custom Managers, Querysets & Manager Inheritance

3.3.1 Using a Custom Manager

A nice feature of Django is the possibility to define one's own custom object managers. This is fully supported with django_polymorphic: For creating a custom polymorphic manager class, just derive your manager from `PolymorphicManager` instead of `models.Manager`. As with vanilla Django, in your model class, you should explicitly add the default manager first, and then your custom manager:

```
from polymorphic.models import PolymorphicModel
from polymorphic.manager import PolymorphicManager

class TimeOrderedManager(PolymorphicManager):
    def get_queryset(self):
        qs = super(TimeOrderedManager,self).get_queryset()
        return qs.order_by('-start_date')         # order the queryset

    def most_recent(self):
        qs = self.get_queryset()                  # get my ordered queryset
        return qs[:10]                            # limit => get ten most recent entries

class Project(PolymorphicModel):
    objects = PolymorphicManager()                # add the default polymorphic manager first
    objects_ordered = TimeOrderedManager()        # then add your own manager
    start_date = DateTimeField()                  # project start is this date/time
```

The first manager defined ('objects' in the example) is used by Django as automatic manager for several purposes, including accessing related objects. It must not filter objects and it's safest to use the plain `PolymorphicManager` here.

> Note that get_query_set is deprecated in Django 1.8 and creates warnings in Django 1.7.

3.3.2 Manager Inheritance

Polymorphic models inherit/propagate all managers from their base models, as long as these are polymorphic. This means that all managers defined in polymorphic base models continue to work as expected in models inheriting from this base model:

```
from polymorphic.models import PolymorphicModel
from polymorphic.manager import PolymorphicManager

class TimeOrderedManager(PolymorphicManager):
    def get_queryset(self):
        qs = super(TimeOrderedManager,self).get_queryset()
        return qs.order_by('-start_date')         # order the queryset

    def most_recent(self):
        qs = self.get_queryset()                  # get my ordered queryset
        return qs[:10]                            # limit => get ten most recent entries
```

```
class Project(PolymorphicModel):
    objects = PolymorphicManager()                # add the default polymorphic manager first
    objects_ordered = TimeOrderedManager()        # then add your own manager
    start_date = DateTimeField()                  # project start is this date/time

class ArtProject(Project):                        # inherit from Project, inheriting its fields and ma
    artist = models.CharField(max_length=30)
```

ArtProject inherited the managers `objects` and `objects_ordered` from Project.

`ArtProject.objects_ordered.all()` will return all art projects ordered regarding their start time and `ArtProject.objects_ordered.most_recent()` will return the ten most recent art projects. .

Note that get_query_set is deprecated in Django 1.8 and creates warnings in Django 1.7.

3.3.3 Using a Custom Queryset Class

The `PolymorphicManager` class accepts one initialization argument, which is the queryset class the manager should use. Just as with vanilla Django, you may define your own custom queryset classes. Just use PolymorphicQuerySet instead of Django's QuerySet as the base class:

```
from polymorphic.models import PolymorphicModel
from polymorphic.manager import PolymorphicManager
from polymorphic.query import PolymorphicQuerySet

class MyQuerySet(PolymorphicQuerySet):
    def my_queryset_method(...):
        ...

class MyModel(PolymorphicModel):
    my_objects=PolymorphicManager(MyQuerySet)
    ...
```

3.4 Third-party applications support

3.4.1 django-reversion support

Support for django-reversion works as expected with polymorphic models. However, they require more setup than standard models. That's become:

- The children models are not registered in the admin site. You will therefore need to manually register them to django-reversion.
- Polymorphic models use multi-table inheritance. See the reversion documentation how to deal with this by adding a `follow` field for the primary key.
- Both admin classes redefine `object_history_template`.

Example

The admin *Example* becomes:

```
from django.contrib import admin
from polymorphic.admin import PolymorphicParentModelAdmin, PolymorphicChildModelAdmin
from reversion.admin import VersionAdmin
from reversion import revisions
from .models import ModelA, ModelB, ModelC


class ModelAChildAdmin(PolymorphicChildModelAdmin, VersionAdmin):
    base_model = ModelA
    base_form = ...
    base_fieldsets = (
        ...
    )

class ModelBAdmin(ModelAChildAdmin, VersionAdmin):
    # define custom features here

class ModelCAdmin(ModelBAdmin):
    # define custom features here


class ModelAParentAdmin(VersionAdmin, PolymorphicParentModelAdmin):
    base_model = ModelA
    child_models = (
        (ModelB, ModelBAdmin),
        (ModelC, ModelCAdmin),
    )

revisions.register(ModelB, follow=['modela_ptr'])
revisions.register(ModelC, follow=['modelb_ptr'])
admin.site.register(ModelA, ModelAParentAdmin)
```

Redefine a `admin/polymorphic/object_history.html` template, so it combines both worlds:

```
{% extends 'reversion/object_history.html' %}
{% load polymorphic_admin_tags %}

{% block breadcrumbs %}
    {% breadcrumb_scope base_opts %}{{ block.super }}{% endbreadcrumb_scope %}
{% endblock %}
```

This makes sure both the reversion template is used, and the breadcrumb is corrected for the polymorphic model.

3.4.2 django-reversion-compare support

The django-reversion-compare views work as expected, the admin requires a little tweak. In your parent admin, include the following method:

```
def compare_view(self, request, object_id, extra_context=None):
    """Redirect the reversion-compare view to the child admin."""
    real_admin = self._get_real_admin(object_id)
    return real_admin.compare_view(request, object_id, extra_context=extra_context)
```

As the compare view resolves the the parent admin, it uses it's base model to find revisions. This doesn't work, since it needs to look for revisions of the child model. Using this tweak, the view of the actual child model is used, similar to the way the regular change and delete views are redirected.

3.4.3 django-mptt support

Combining polymorphic with django-mptt is certainly possible, but not straightforward. It involves combining both managers, querysets, models, meta-classes and admin classes using multiple inheritance.

The django-polymorphic-tree package provides this out of the box.

3.5 Changelog

3.5.1 Version 0.9.2 (2016-05-04)

- Fix error when using `date_hierarchy` field in the admin
- Fixed Django 1.10 warning in admin add-type view.

3.5.2 Version 0.9.1 (2016-02-18)

- Fixed support for `PolymorphicManager.from_queryset()` for custom query sets.
- Fixed Django 1.7 `changeform_view()` redirection to the child admin site. This fixes custom admin code that uses these views, such as django-reversion's `revision_view()` / `recover_view()`.
- Fixed `.only('pk')` field support.
- Fixed `object_history_template` breadcrumb. **NOTE:** when using django-reversion / django-reversion-compare, make sure to implement a `admin/polymorphic/object_history.html` template in your project that extends from `reversion/object_history.html` or `reversion-compare/object_history.html` respectively.

3.5.3 Version 0.9 (2016-02-17)

- Added `.only()` and `.defer()` support.
- Added support for Django 1.8 complex expressions in `.annotate()` / `.aggregate()`.
- Fix Django 1.9 handling of custom URLs. The new change-URL redirect overlapped any custom URLs defined in the child admin.
- Fix Django 1.9 support in the admin.
- Fix missing `history_view()` redirection to the child admin, which is important for django-reversion support. See the documentation for hints for *django-reversion-compare support*.

3.5.4 Version 0.8.1 (2015-12-29)

- Fixed support for reverse relations for `relname___field` when the field starts with an _ character. Otherwise, the query will be interpreted as subclass lookup (`ClassName___field`).

3.5.5 Version 0.8 (2015-12-28)

- Added Django 1.9 compatibility.
- Renamed `polymorphic.manager` => `polymorphic.managers` for consistentcy.

- **BACKWARDS INCOMPATIBILITY:** The import paths have changed to support Django 1.9. Instead of `from polymorphic import X`, you'll have to import from the proper package. For example:

```
from polymorphic.models import PolymorphicModel
from polymorphic.managers import PolymorphicManager, PolymorphicQuerySet
from polymorphic.showfields import ShowFieldContent, ShowFieldType, ShowFieldTypeAndContent
```

- **BACKWARDS INCOMPATIBILITY:** Removed `__version__.py` in favor of a standard `__version__` in `polymorphic/__init__.py`.
- **BACKWARDS INCOMPATIBILITY:** Removed automatic proxying of method calls to the queryset class. Use the standard Django methods instead:

```
# In model code:
objects = PolymorphicQuerySet.as_manager()

# For manager code:
MyCustomManager = PolymorphicManager.from_queryset(MyCustomQuerySet)
```

3.5.6 Version 0.7.2 (2015-10-01)

- Added `queryset.as_manager()` support for Django 1.7/1.8
- Optimize model access for non-dumpdata usage; avoid `__getattribute__()` call each time to access the manager.
- Fixed 500 error when using invalid PK's in the admin URL, return 404 instead.
- Fixed possible issues when using an custom `AdminSite` class for the parent object.
- Fixed Pickle exception when polymorphic model is cached.

3.5.7 Version 0.7.1 (2015-04-30)

- Fixed Django 1.8 support for related field widgets.

3.5.8 Version 0.7 (2015-04-08)

- Added Django 1.8 support
- Added support for custom primary key defined using `mybase_ptr = models.OneToOneField(BaseClass, parent_link=True, related_name="...")`.
- Fixed Python 3 issue in the admin
- Fixed `_default_manager` to be consistent with Django, it's now assigned directly instead of using `add_to_class()`
- Fixed 500 error for admin URLs without a '/', e.g. `admin/app/parentmodel/id`.
- Fixed preserved filter for Django admin in delete views
- Removed test noise for diamond inheritance problem (which Django 1.7 detects)

3.5.9 Version 0.6.1 (2014-12-30)

- Remove Django 1.7 warnings
- Fix Django 1.4/1.5 queryset calls on related objects for unknown methods. The `RelatedManager` code overrides `get_query_set()` while `__getattr__()` used the new-style `get_queryset()`.
- Fix validate_model_fields(), caused errors when metaclass raises errors

3.5.10 Version 0.6 (2014-10-14)

- Added Django 1.7 support.
- Added permission check for all child types.
- **BACKWARDS INCOMPATIBILITY:** the `get_child_type_choices()` method receives 2 arguments now (request, action). If you have overwritten this method in your code, make sure the method signature is updated accordingly.

3.5.11 Version 0.5.6 (2014-07-21)

- Added `pk_regex` to the `PolymorphicParentModelAdmin` to support non-integer primary keys.
- Fixed passing `?ct_id=` to the add view for Django 1.6 (fixes compatibility with django-parler).

3.5.12 Version 0.5.5 (2014-04-29)

- Fixed `get_real_instance_class()` for proxy models (broke in 0.5.4).

3.5.13 Version 0.5.4 (2014-04-09)

- Fix `.non_polymorphic()` to returns a clone of the queryset, instead of effecting the existing queryset.
- Fix missing `alters_data = True` annotations on the overwritten `save()` methods.
- Fix infinite recursion bug in the admin with Django 1.6+
- Added detection of bad `ContentType` table data.

3.5.14 Version 0.5.3 (2013-09-17)

- Fix TypeError when `base_form` was not defined.
- Fix passing `/admin/app/model/id/XYZ` urls to the correct admin backend. There is no need to include a `?ct_id=..` field, as the ID already provides enough information.

3.5.15 Version 0.5.2 (2013-09-05)

- Fix Grappelli breadcrumb support in the views.
- Fix unwanted `___` handling in the ORM when a field name starts with an underscore; this detects you meant `relatedfield__ _underscorefield` instead of `ClassName___field`.
- Fix missing permission check in the "add type" view. This was caught however in the next step.

- Fix admin validation errors related to additional non-model form fields.

3.5.16 Version 0.5.1 (2013-07-05)

- Add Django 1.6 support.
- Fix Grappelli theme support in the "Add type" view.

3.5.17 Version 0.5 (2013-04-20)

- Add Python 3.2 and 3.3 support
- Fix errors with ContentType objects that don't refer to an existing model.

3.5.18 Version 0.4.2 (2013-04-10)

- Used proper `__version__` marker.

3.5.19 Version 0.4.1 (2013-04-10)

- Add Django 1.5 and 1.6 support
- Add proxy model support
- Add default admin `list_filter` for polymorphic model type.
- Fix queryset support of related objects.
- Performed an overall cleanup of the project
- **Deprecated** the `queryset_class` argument of the `PolymorphicManager` constructor, use the class attribute instead.
- **Dropped** Django 1.1, 1.2 and 1.3 support

3.5.20 Version 0.4 (2013-03-25)

- Update example project for Django 1.4
- Added tox and Travis configuration

3.5.21 Version 0.3.1 (2013-02-28)

- SQL optimization, avoid query in pre_save_polymorphic()

3.5.22 Version 0.3 (2013-02-28)

Many changes to the codebase happened, but no new version was released to pypi for years. 0.3 contains fixes submitted by many contributors, huge thanks to everyone!

- Added a polymorphic admin interface.
- PEP8 and code cleanups by various authors

3.5.23 Version 0.2 (2011-04-27)

The 0.2 release serves as legacy release. It supports Django 1.1 up till 1.4 and Python 2.4 up till 2.7.

For a detailed list of it's changes, see the `archived changelog`.

3.6 Contributing

You can contribute to *django-polymorphic* to forking the code on GitHub:

https://github.com/chrisglass/django_polymorphic

3.6.1 Running tests

We require features to be backed by a unit test. This way, we can test *django-polymorphic* against new Django versions. To run the included test suite, execute:

```
./runtests.py
```

To test support for multiple Python and Django versions, run tox from the repository root:

```
pip install tox
tox
```

The Python versions need to be installed at your system. On Linux, download the versions at http://www.python.org/download/releases/. On MacOS X, use Homebrew to install other Python versions.

We currently support Python 2.6, 2.7, 3.2 and 3.3.

3.6.2 Example project

The repository contains a complete Django project that may be used for tests or experiments, without any installation needed.

The management command `pcmd.py` in the app `pexp` can be used for quick tests or experiments - modify this file (pexp/management/commands/pcmd.py) to your liking.

3.6.3 Supported Django versions

The current release should be usable with the supported releases of Django; the current stable release and the previous release. Supporting older Django versions is a nice-to-have feature, but not mandatory.

In case you need to use *django-polymorphic* with older Django versions, consider installing a previous version.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search