django-polymorphic Documentation

Release 2.0.2

Bert Constantin, Chris Glass, Diederik van der Boor

Feb 05, 2018

Contents

Django-polymorphic builds on top of the standard Django model inheritance. It makes using inherited models easier. 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.
- Formset support.
- 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 1.11, 2.0 and Python 2.7 and 3.4+ is supported. For older Django versions, use *django-polymorphic==1.3*.

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. Turner'))
[ <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.

2.2.1 Setup

Both the parent model and child model need to have a `ModelAdmin` class.

The shared base model should use the *`PolymorphicParentModelAdmin`* as base class.

- *`base_model`* should be set
- *`child_models`* or *`get_child_models()`* should return an iterable of Model classes.

The admin class for every child model should inherit from *`PolymorphicChildModelAdmin`*

- *`base_model`* should be set.

Although the child models are registered too, they won't be shown in the admin index page. This only happens when *`show_in_index`* is set to `True`.

Fieldset configuration

The parent admin is only used for the list display of models, and for the edit/delete view of non-subclassed models.

All other model types are redirected to the edit/delete/history view of the child model admin. Hence, the fieldset configuration should be placed on the child admin.

Tip: When the child admin is used as base class for various derived classes, avoid using the standard `ModelAdmin` attributes `form` and `fieldsets`. Instead, use the `base_form` and `base_fieldsets` attributes. This allows the *`PolymorphicChildModelAdmin`* class to detect any additional fields in case the child model is overwritten.

Changed in version 1.0: It's now needed to register the child model classes too.

In *django-polymorphic* 0.9 and below, the `child_models` was a tuple of a `(Model, ChildModelAdmin)`. The admin classes were registered in an internal class, and kept away from the main admin site. This caused various subtle problems with the `ManyToManyField` and related field wrappers, which are fixed by registering the child admin classes too. Note that they are hidden from the main view, unless *`show_in_index`* is set.

2.2.2 Example

The models are taken from *Advanced features*.

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


class ModelAChildAdmin(PolymorphicChildModelAdmin):
    """ Base admin class for all child models """
    base_model = ModelA  # Optional, explicitly set here.

    # 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 = (
        ...
    )
```

```
@admin.register(ModelB)
class ModelBAdmin(ModelAChildAdmin):
    base_model = ModelB  # Explicitly set here!
    # define custom features here


@admin.register(ModelC)
class ModelCAdmin(ModelBAdmin):
    base_model = ModelC  # Explicitly set here!
    show_in_index = True  # makes child model admin visible in main admin site
    # define custom features here


@admin.register(ModelA)
class ModelAParentAdmin(PolymorphicParentModelAdmin):
    """ The parent model admin """
    base_model = ModelA  # Optional, explicitly set here.
    child_models = (ModelB, ModelC)
    list_filter = (PolymorphicChildModelFilter,)  # This is optional.
```

2.2.3 Filtering child types

Child model types can be filtered by adding a *PolymorphicChildModelFilter* to the `list_filter` attribute. See the example above.

2.2.4 Inline models

New in version 1.0.

Inline models are handled via a special *StackedPolymorphicInline* class.

For models with a generic foreign key, there is a *GenericStackedPolymorphicInline* class available.

When the inline is included to a normal `ModelAdmin`, make sure the *PolymorphicInlineSupportMixin* is included. This is not needed when the admin inherits from the *PolymorphicParentModelAdmin* / *PolymorphicChildModelAdmin* classes.

In the following example, the `PaymentInline` supports several types. These are defined as separate inline classes. The child classes can be nested for clarity, but this is not a requirement.

```
from django.contrib import admin

from polymorphic.admin import PolymorphicInlineSupportMixin, StackedPolymorphicInline
from .models import Order, Payment, CreditCardPayment, BankPayment, SepaPayment


class PaymentInline(StackedPolymorphicInline):
    """
    An inline for a polymorphic model.
    The actual form appearance of each row is determined by
    the child inline that corresponds with the actual model type.
    """
    class CreditCardPaymentInline(StackedPolymorphicInline.Child):
        model = CreditCardPayment
```

```
    class BankPaymentInline(StackedPolymorphicInline.Child):
        model = BankPayment

    class SepaPaymentInline(StackedPolymorphicInline.Child):
        model = SepaPayment

    model = Payment
    child_inlines = (
        CreditCardPaymentInline,
        BankPaymentInline,
        SepaPaymentInline,
    )


@admin.register(Order)
class OrderAdmin(PolymorphicInlineSupportMixin, admin.ModelAdmin):
    """
    Admin for orders.
    The inline is polymorphic.
    To make sure the inlines are properly handled,
    the ``PolymorphicInlineSupportMixin`` is needed to
    """
    inlines = (PaymentInline,)
```

Using polymorphic models in standard 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".

```
from django.contrib import admin
from .models import StandardModel


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


@admin.register(StandardModel)
class StandardModelAdmin(admin.ModelAdmin):
    inlines = [ModelBInline]
```

2.2.5 Internal details

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 redirects the *edit* and *delete* views to the `ModelAdmin` of the derived child model. The *list* page is still implemented by the parent model admin.

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 an iterable of Model classes.

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.

If you use other applications such as django-reversion or django-mptt, please check +:ref:*third-party*.

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.

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.
- Although it must be registered with admin site, by default it's hidden from admin site index page. This can be overriden by adding *`show_in_index`* = `True` in admin class.

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.

2.4 Third-party applications support

2.4.1 django-guardian support

New in version 1.0.2.

You can configure django-guardian to use the base model for object level permissions. Add this option to your settings:

```
GUARDIAN_GET_CONTENT_TYPE = 'polymorphic.contrib.guardian.get_polymorphic_base_
↪content_type'
```

This option requires django-guardian >= 1.4.6. Details about how this option works are available in the django-guardian documentation.

2.4.2 django-extra-views

New in version 1.1.

The *polymorphic.contrib.extra_views* package provides classes to display polymorphic formsets using the classes from django-extra-views. See the documentation of:

- *PolymorphicFormSetView*
- *PolymorphicInlineFormSetView*
- *PolymorphicInlineFormSet*

2.4.3 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.

2.4.4 django-reversion support

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

- Manually register the child models with django-reversion, so their `follow` parameter can be set.
- 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 *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  # optional, explicitly set here.
    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  # optional, explicitly set here.
    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.

2.4.5 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.

CHAPTER 3

Advanced topics

3.1 Formsets

New in version 1.0.

Polymorphic models can be used in formsets.

The implementation is almost identical to the regular Django formsets. As extra parameter, the factory needs to know how to display the child models. Provide a list of *PolymorphicFormSetChild* objects for this.

```
from polymorphic.formsets import polymorphic_modelformset_factory,␣
↪PolymorphicFormSetChild

ModelAFormSet = polymorphic_modelformset_factory(ModelA, formset_children=(
    PolymorphicFormSetChild(ModelB),
    PolymorphicFormSetChild(ModelC),
))
```

The formset can be used just like all other formsets:

```
if request.method == "POST":
    formset = ModelAFormSet(request.POST, request.FILES, queryset=ModelA.objects.
↪all())
    if formset.is_valid():
        formset.save()
else:
    formset = ModelAFormSet(queryset=ModelA.objects.all())
```

Like standard Django formsets, there are 3 factory methods available:

- *polymorphic_modelformset_factory()* - create a regular model formset.
- *polymorphic_inlineformset_factory()* - create a inline model formset.
- *generic_polymorphic_inlineformset_factory()* - create an inline formset for a generic foreign key.

Each one uses a different base class:

- *BasePolymorphicModelFormSet*
- *BasePolymorphicInlineFormSet*
- *BaseGenericPolymorphicInlineFormSet*

When needed, the base class can be overwritten and provided to the factory via the `formset` parameter.

3.2 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.

3.2.1 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)


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='contenttypes.ContentType', null=True),
        ),
        migrations.RunPython(forwards_func, migrations.RunPython.noop),
    ]
```

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

New in version 1.1.

When the model is created elsewhere, you can also use the *polymorphic.utils.reset_polymorphic_ctype()* function:

```
from polymorphic.utils import reset_polymorphic_ctype
from myapp.models import Base, Sub1, Sub2

reset_polymorphic_ctype(Base, Sub1, Sub2)

reset_polymorphic_ctype(Base, Sub1, Sub2, ignore_existing=True)
```

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.

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 managers
    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.

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(self):
        ...

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

3.4 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)
```

3.4.1 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) )
```

3.4.2 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)> ]
```

3.4.3 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)> ]
```

3.4.4 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)> ]
```

3.4.5 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
```

3.4.6 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)> ]
```

3.4.7 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.

3.4.8 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.

3.4.9 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.models import PolymorphicModel
from polymorphic.showfields import 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`.

3.4.10 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.

3.5 Changelog

3.5.1 Changes in 2.0.2 (2018-02-05)

- Fixed manager inheritance behavior for Django 1.11, by automatically enabling `Meta.manager_inheritance_from_future` if it's not defined. This restores the manager inheritance behavior that *django-polymorphic 1.3* provided for Django 1.x projects.
- Fixed internal `base_objects` usage.

3.5.2 Changes in 2.0.1 (2018-02-05)

- Fixed manager inheritance detection for Django 1.11.

  It's recommended to use `Meta.manager_inheritance_from_future` so Django 1.x code also inherit the `PolymorphicManager` in all subclasses. Django 2.0 already does this by default.
- Deprecated the `base_objects` manager. Use `objects.non_polymorphic()` instead.
- Optimized detection for dumpdata behavior, avoiding the performance hit of `__getattribute__()`.
- Fixed test management commands

3.5.3 Changes in 2.0 (2018-01-22)

- **BACKWARDS INCOMPATIBILITY:** Dropped Django 1.8 and 1.10 support.
- **BACKWARDS INCOMPATIBILITY:** Removed old deprecated code from 1.0, thus:
- Import managers from `polymorphic.managers` (plural), not `polymorphic.manager`.
- Register child models to the admin as well using `@admin.register()` or `admin.site.register()`, as this is no longer done automatically.
- Added Django 2.0 support.
- Added `PolymorphicTypeUndefined` exception for incomplete imported models. When a data migration or import creates an polymorphic model, the `polymorphic_ctype_id` field should be filled in manually too. The `polymorphic.utils.reset_polymorphic_ctype` function can be used for that.
- Added `PolymorphicTypeInvalid` exception when database was incorrectly imported.
- Added `polymorphic.utils.get_base_polymorphic_model()` to find the base model for types.
- Using `base_model` on the polymorphic admins is no longer required, as this can be autodetected.
- Fixed manager errors for swappable models.
- Fixed `deleteText` of `|as_script_options` template filter.
- Fixed `.filter(applabel__ModelName___field=...)` lookups.
- Improved `polymorphic.utils.reset_polymorphic_ctype()` to accept models in random ordering.
- Fix fieldsets handling in the admin (`declared_fieldsets` is removed since Django 1.9)

3.5.4 Version 1.3 (2017-08-01)

- **BACKWARDS INCOMPATIBILITY:** Dropped Django 1.4, 1.5, 1.6, 1.7, 1.9 and Python 2.6 support. Only official Django releases (1.8, 1.10, 1.11) are supported now.
- Allow expressions to pass unchanged in `.order_by()`
- Fixed Django 1.11 accessor checks (to support subclasses of `ForwardManyToOneDescriptor`, like `ForwardOneToOneDescriptor`)
- Fixed polib syntax error messages in translations.

3.5.5 Version 1.2 (2017-05-01)

- Django 1.11 support.
- Fixed `PolymorphicInlineModelAdmin` to explictly exclude `polymorphic_ctype`.
- Fixed Python 3 TypeError in the admin when preserving the query string.
- Fixed Python 3 issue due to `force_unicode()` usage instead of `force_text()`.
- Fixed `z-index` attribute for admin menu appearance.

3.5.6 Version 1.1 (2017-02-03)

- Added class based formset views in `polymorphic/contrib/extra_views`.
- Added helper function `polymorphic.utils.reset_polymorphic_ctype()`. This eases the migration old existing models to polymorphic.
- Fixed Python 2.6 issue.
- Fixed Django 1.6 support.

3.5.7 Version 1.0.2 (2016-10-14)

- Added helper function for django-guardian; add `GUARDIAN_GET_CONTENT_TYPE = 'polymorphic.contrib.guardian.get_polymorphic_base_content_type'` to the project settings to let guardian handles inherited models properly.
- Fixed `polymorphic_modelformset_factory()` usage.
- Fixed Python 3 bug for inline formsets.
- Fixed CSS for Grappelli, so model choice menu properly overlaps.
- Fixed `ParentAdminNotRegistered` exception for models that are registered via a proxy model instead of the real base model.

3.5.8 Version 1.0.1 (2016-09-11)

- Fixed compatibility with manager changes in Django 1.10.1

3.5.9 Version 1.0 (2016-09-02)

- Added Django 1.10 support.
- Added **admin inline** support for polymorphic models.
- Added **formset** support for polymorphic models.
- Added support for polymorphic queryset limiting effects on *proxy models*.
- Added support for multiple databases with the `.using()` method and `using=..` keyword argument.
- Fixed modifying passed `Q()` objects in place.

Note: This version provides a new method for registering the admin models. While the old method is still supported, we recommend to upgrade your code. The new registration style improves the compatibility in the Django admin.

- Register each `PolymorphicChildModelAdmin` with the admin site too.
- The `child_models` attribute of the `PolymorphicParentModelAdmin` should be a flat list of all child models. The `(model, admin)` tuple is obsolete.

Also note that proxy models will now limit the queryset too.

Fixed since 1.0b1 (2016-08-10)

- Fix formset empty-form display when there are form errors.
- Fix formset empty-form hiding for Grappelli.
- Fixed packing `admin/polymorphic/edit_inline/stacked.html` in the wheel format.

3.5.10 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.

3.5.11 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.

3.5.12 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 setting an extra custom manager without overriding the `_default_manager`.
- 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*.

3.5.13 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`).

3.5.14 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)
```

3.5.15 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.

3.5.16 Version 0.7.1 (2015-04-30)

- Fixed Django 1.8 support for related field widgets.

3.5.17 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)

3.5.18 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

3.5.19 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.

3.5.20 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).

3.5.21 Version 0.5.5 (2014-04-29)

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

3.5.22 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.

3.5.23 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.

3.5.24 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.

3.5.25 Version 0.5.1 (2013-07-05)

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

3.5.26 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.

3.5.27 Version 0.4.2 (2013-04-10)

- Used proper `__version__` marker.

3.5.28 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

3.5.29 Version 0.4 (2013-03-25)

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

3.5.30 Version 0.3.1 (2013-02-28)

- SQL optimization, avoid query in pre_save_polymorphic()

3.5.31 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

3.5.32 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/django-polymorphic/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.

3.7 API Documentation

3.7.1 polymorphic.admin

ModelAdmin classes

The `PolymorphicParentModelAdmin` class

class polymorphic.admin.**PolymorphicParentModelAdmin**(*model*, *admin_site*, **args*, ***kwargs*)

Bases: `django.contrib.admin.options.ModelAdmin`

A admin interface that can displays different change/delete pages, depending on the polymorphic model. To use this class, one attribute need to be defined:

- *child_models* should be a list models.

Alternatively, the following methods can be implemented:

- *get_child_models()* should return a list of models.
- optionally, *get_child_type_choices()* can be overwritten to refine the choices for the add dialog.

This class needs to be inherited by the model admin base class that is registered in the site. The derived models should *not* register the ModelAdmin, but instead it should be returned by *get_child_models()*.

add_type_form
alias of *PolymorphicModelChoiceForm*

add_type_view(*request*, *form_url=''*)
Display a choice form to select which page type to add.

add_view(*request*, *form_url=''*, *extra_context=None*)
Redirect the add view to the real admin.

change_view(*request*, *object_id*, **args*, ***kwargs*)
Redirect the change view to the real admin.

changeform_view(*request*, *object_id=None*, **args*, ***kwargs*)

delete_view(*request*, *object_id*, *extra_context=None*)
Redirect the delete view to the real admin.

get_child_models()
Return the derived model classes which this admin should handle. This should return a list of tuples, exactly like *child_models* is.

The model classes can be retrieved as `base_model.__subclasses__()`, a setting in a config file, or a query of a plugin registration system at your option

get_child_type_choices(*request*, *action*)
Return a list of polymorphic types for which the user has the permission to perform the given action.

get_preserved_filters(*request*)

get_queryset(*request*)

get_urls()
Expose the custom URLs for the subclasses and the URL resolver.

history_view(*request*, *object_id*, *extra_context=None*)
Redirect the history view to the real admin.

register_child(*model*, *model_admin*)
Register a model with admin to display.

render_add_type_form(*request*, *context*, *form_url=”*)
Render the page type choice form.

subclass_view(*request*, *path*)
Forward any request to a custom view of the real admin.

add_type_template = None

base_model = None
The base model that the class uses (auto-detected if not set explicitly)

change_list_template

child_models = None
The child models that should be displayed

media

pk_regex = '(\\d+|__fk__)'
The regular expression to filter the primary key in the URL. This accepts only numbers as defensive measure against catch-all URLs. If your primary key consists of string values, update this regular expression.

polymorphic_list = False
Whether the list should be polymorphic too, leave to `False` to optimize

The `PolymorphicChildModelAdmin` class

class polymorphic.admin.**PolymorphicChildModelAdmin**(*model*, *admin_site*, **args*, ***kwargs*)
Bases: `django.contrib.admin.options.ModelAdmin`

The *optional* base class for the admin interface of derived models.

This base class defines some convenience behavior for the admin interface:

- It corrects the breadcrumbs in the admin pages.
- It adds the base model to the template lookup paths.
- 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.

delete_view(*request*, *object_id*, *context=None*)

get_base_fieldsets(*request*, *obj=None*)

get_fieldsets(*request*, *obj=None*)

get_form(*request*, *obj=None*, ***kwargs*)

get_model_perms(*request*)

get_subclass_fields(*request*, *obj=None*)

history_view(*request*, *object_id*, *extra_context=None*)

render_change_form(*request*, *context*, *add=False*, *change=False*, *form_url=”*, *obj=None*)

response_post_save_add(*request*, *obj*)

response_post_save_change(*request*, *obj*)

base_fieldsets = None
By setting `base_fieldsets` instead of `fieldsets`, any subclass fields can be automatically added. This is useful when your model admin class is inherited by others.

base_form = None
By setting `base_form` instead of `form`, any subclass fields are automatically added to the form. This is useful when your model admin class is inherited by others.

base_model = None
The base model that the class uses (auto-detected if not set explicitly)

change_form_template

delete_confirmation_template

extra_fieldset_title = u'Contents'
Default title for extra fieldset

media

object_history_template

show_in_index = False
Whether the child admin model should be visible in the admin index page.

List filtering

The `PolymorphicChildModelFilter` class

class polymorphic.admin.**PolymorphicChildModelFilter**(*request*, *params*, *model*, *model_admin*)
Bases: `django.contrib.admin.filters.SimpleListFilter`

An admin list filter for the PolymorphicParentModelAdmin which enables filtering by its child models.

This can be used in the parent admin:

```
list_filter = (PolymorphicChildModelFilter,)
```

Inlines support

The `StackedPolymorphicInline` class

class polymorphic.admin.**StackedPolymorphicInline**(*parent_model*, *admin_site*)
Bases: `polymorphic.admin.inlines.PolymorphicInlineModelAdmin`

Stacked inline for django-polymorphic models. Since tabular doesn't make much sense with changed fields, just offer this one.

The `GenericStackedPolymorphicInline` class

class polymorphic.admin.**GenericStackedPolymorphicInline**(*parent_model*, *admin_site*)
Bases: `polymorphic.admin.generic.GenericPolymorphicInlineModelAdmin`

The stacked layout for generic inlines.

media

template = 'admin/polymorphic/edit_inline/stacked.html'
The default template to use.

The PolymorphicInlineSupportMixin class

class polymorphic.admin.**PolymorphicInlineSupportMixin**
Bases: object

A Mixin to add to the regular admin, so it can work with our polymorphic inlines.

This mixin needs to be included in the admin that hosts the inlines. It makes sure the generated admin forms have different fieldsets/fields depending on the polymorphic type of the form instance.

This is achieved by overwriting *get_inline_formsets()* to return an *PolymorphicInlineAdminFormSet* instead of a standard Django InlineAdminFormSet for the polymorphic formsets.

get_inline_formsets(*request*, *formsets*, *inline_instances*, *obj=None*, **args*, ***kwargs*)
Overwritten version to produce the proper admin wrapping for the polymorphic inline formset. This fixes the media and form appearance of the inline polymorphic models.

Low-level classes

These classes are useful when existing parts of the admin classes.

class polymorphic.admin.**PolymorphicModelChoiceForm**(**args*, ***kwargs*)
Bases: django.forms.forms.Form

The default form for the add_type_form. Can be overwritten and replaced.

base_fields = OrderedDict([('ct_id', <django.forms.fields.ChoiceField object>)])

declared_fields = OrderedDict([('ct_id', <django.forms.fields.ChoiceField object>)])

media

type_label = u'Type'
Define the label for the radiofield

class polymorphic.admin.**PolymorphicInlineModelAdmin**(*parent_model*, *admin_site*)
Bases: django.contrib.admin.options.InlineModelAdmin

A polymorphic inline, where each formset row can be a different form.

Note that:

- Permissions are only checked on the base model.
- The child inlines can't override the base model fields, only this parent inline can do that.

class Child(*parent_inline*)
Bases: django.contrib.admin.options.InlineModelAdmin

The child inline; which allows configuring the admin options for the child appearance.

Note that not all options will be honored by the parent, notably the formset options: * *extra* * min_num * max_num

The model form options however, will all be read.

formset_child
alias of PolymorphicFormSetChild

get_fields(*request*, *obj=None*)

get_formset(*request*, *obj=None*, ***kwargs*)

get_formset_child(*request*, *obj=None*, ***kwargs*)
Return the formset child that the parent inline can use to represent us.
Return type *PolymorphicFormSetChild*

extra = 0

media

formset
alias of `BasePolymorphicInlineFormSet`

get_child_inline_instance(*model*)
Find the child inline for a given model.

Return type *PolymorphicInlineModelAdmin.Child*

get_child_inline_instances()
:rtype List[PolymorphicInlineModelAdmin.Child]

get_fields(*request*, *obj=None*)

get_fieldsets(*request*, *obj=None*)
Hook for specifying fieldsets.

get_formset(*request*, *obj=None*, ***kwargs*)
Construct the inline formset class.

This passes all class attributes to the formset.

Return type type

get_formset_children(*request*, *obj=None*)
The formset 'children' provide the details for all child models that are part of this formset. It provides a stripped version of the modelform/formset factory methods.

child_inlines = ()
Inlines for all model sub types that can be displayed in this inline. Each row is a *PolymorphicInlineModelAdmin.Child*

extra = 0
The extra forms to show By default there are no 'extra' forms as the desired type is unknown. Instead, add each new item using JavaScript that first offers a type-selection.

media

polymorphic_media = <django.forms.widgets.Media object>
The extra media to add for the polymorphic inlines effect. This can be redefined for subclasses.

class polymorphic.admin.**GenericPolymorphicInlineModelAdmin**(*parent_model*, *admin_site*)
Bases: `polymorphic.admin.inlines.PolymorphicInlineModelAdmin`, `django.contrib.contenttypes.admin.GenericInlineModelAdmin`

Base class for variation of inlines based on generic foreign keys.

class Child(*parent_inline*)
Bases: `polymorphic.admin.inlines.Child`

Variation for generic inlines.

formset_child
alias of `GenericPolymorphicFormSetChild`

get_formset_child(*request*, *obj=None*, ***kwargs*)

content_type
Expose the ContentType that the child relates to. This can be used for the `polymorphic_ctype` field.

ct_field = 'content_type'

ct_fk_field = 'object_id'

media

formset
The formset class

alias of `BaseGenericPolymorphicInlineFormSet`

get_formset(*request*, *obj=None*, ***kwargs*)
Construct the generic inline formset class.

media

class polymorphic.admin.**PolymorphicInlineAdminForm**(*formset*, *form*, *fieldsets*, *prepopulated_fields*, *original*, *readonly_fields=None*, *model_admin=None*, *view_on_site_url=None*)
Bases: `django.contrib.admin.helpers.InlineAdminForm`

Expose the admin configuration for a form

class polymorphic.admin.**PolymorphicInlineAdminFormSet**(**args*, ***kwargs*)
Bases: `django.contrib.admin.helpers.InlineAdminFormSet`

Internally used class to expose the formset in the template.

3.7.2 polymorphic.contrib.extra_views

The `extra_views.formsets` provides a simple way to handle formsets. The `extra_views.advanced` provides a method to combine that with a create/update form.

This package provides classes that support both options for polymorphic formsets.

class polymorphic.contrib.extra_views.**PolymorphicFormSetView**(***kwargs*)
Bases: `polymorphic.contrib.extra_views.PolymorphicFormSetMixin`, `extra_views.formsets.ModelFormSetView`

A view that displays a single polymorphic formset.

```
from polymorphic.formsets import PolymorphicFormSetChild


class ItemsView(PolymorphicFormSetView):
    model = Item
    formset_children = [
        PolymorphicFormSetChild(ItemSubclass1),
        PolymorphicFormSetChild(ItemSubclass2),
    ]
```

formset_class
alias of `BasePolymorphicModelFormSet`

class polymorphic.contrib.extra_views.**PolymorphicInlineFormSetView**(***kwargs*)

Bases: polymorphic.contrib.extra_views.PolymorphicFormSetMixin, extra_views.formsets.InlineFormSetView

A view that displays a single polymorphic formset - with one parent object. This is a variation of the extra_views package classes for django-polymorphic.

```
from polymorphic.formsets import PolymorphicFormSetChild


class OrderItemsView(PolymorphicInlineFormSetView):
    model = Order
    inline_model = Item
    formset_children = [
        PolymorphicFormSetChild(ItemSubclass1),
        PolymorphicFormSetChild(ItemSubclass2),
    ]
```

formset_class

alias of BasePolymorphicInlineFormSet

class polymorphic.contrib.extra_views.**PolymorphicInlineFormSet**(*parent_model*, *request*, *instance*, *view_kwargs=None*, *view=None*)

Bases: polymorphic.contrib.extra_views.PolymorphicFormSetMixin, extra_views.advanced.InlineFormSet

An inline to add to the inlines of the CreateWithInlinesView and UpdateWithInlinesView class.

```
from polymorphic.formsets import PolymorphicFormSetChild


class ItemsInline(PolymorphicInlineFormSet):
    model = Item
    formset_children = [
        PolymorphicFormSetChild(ItemSubclass1),
        PolymorphicFormSetChild(ItemSubclass2),
    ]


class OrderCreateView(CreateWithInlinesView):
    model = Order
    inlines = [ItemsInline]

    def get_success_url(self):
        return self.object.get_absolute_url()
```

formset_class

alias of BasePolymorphicInlineFormSet

3.7.3 polymorphic.contrib.guardian

polymorphic.contrib.guardian.**get_polymorphic_base_content_type**(*obj*)

Helper function to return the base polymorphic content type id. This should used with django-guardian and the GUARDIAN_GET_CONTENT_TYPE option.

See the django-guardian documentation for more information:

https://django-guardian.readthedocs.io/en/latest/configuration.html#guardian-get-content-type

3.7.4 polymorphic.formsets

This allows creating formsets where each row can be a different form type. The logic of the formsets work similar to the standard Django formsets; there are factory methods to construct the classes with the proper form settings.

The "parent" formset hosts the entire model and their child model. For every child type, there is an `PolymorphicFormSetChild` instance that describes how to display and construct the child. It's parameters are very similar to the parent's factory method.

Model formsets

polymorphic.formsets.**polymorphic_modelformset_factory**(*model*, *formset_children*, *formset=<class 'polymorphic.formsets.models.BasePolymorphicModelFormSet'>*, *form=<class 'django.forms.models.ModelForm'>*, *fields=None*, *exclude=None*, *extra=1*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*, *child_form_kwargs=None*)

Construct the class for an polymorphic model formset.

All arguments are identical to :func:'~django.forms.models.modelformset_factory', with the exception of the ''formset_children" argument.

Parameters **formset_children** (*Iterable[PolymorphicFormSetChild]*) – A list of all child :class:'PolymorphicFormSetChild' objects that tell the inline how to render the child model types.

Return type type

class polymorphic.formsets.**PolymorphicFormSetChild**(*model*, *form=<class 'django.forms.models.ModelForm'>*, *fields=None*, *exclude=None*, *formfield_callback=None*, *widgets=None*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*)

Metadata to define the inline of a polymorphic child. Provide this information in the :func:'polymorphic_inlineformset_factory' construction.

Inline formsets

polymorphic.formsets.**polymorphic_inlineformset_factory**(*parent_model*, *model*, *formset_children*, *formset=<class 'polymorphic.formsets.models.BasePolymorphicInlineFormSet'>*, *fk_name=None*, *form=<class 'django.forms.models.ModelForm'>*, *fields=None*, *exclude=None*, *extra=1*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*, *child_form_kwargs=None*)

Construct the class for an inline polymorphic formset.

All arguments are identical to :func:'~django.forms.models.inlineformset_factory', with the exception of the ''formset_children" argument.

Parameters **formset_children** (*Iterable*[PolymorphicFormSetChild]) – A list of all child :class:'PolymorphicFormSetChild' objects that tell the inline how to render the child model types.

Return type type

Generic formsets

polymorphic.formsets.**generic_polymorphic_inlineformset_factory**(*model*, *formset_children*, *form=<class 'django.forms.models.ModelForm'>*, *formset=<class 'polymorphic.formsets.generic.BaseGenericPolym ct_field='content_type'*, *fk_field='object_id'*, *fields=None*, *exclude=None*, *extra=1*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *validate_max=False*, *for_concrete_model=True*, *min_num=None*, *validate_min=False*, *child_form_kwargs=None*)

Construct the class for a generic inline polymorphic formset.

All arguments are identical to `generic_inlineformset_factory()`, with the exception of the `formset_children` argument.

Parameters **formset_children** (*Iterable[PolymorphicFormSetChild]*) – A list of all child *PolymorphicFormSetChild* objects that tell the inline how to render the child model types.

Return type type

Low-level features

The internal machinery can be used to extend the formset classes. This includes:

polymorphic.formsets.**polymorphic_child_forms_factory**(*formset_children*, ***kwargs*)

Construct the forms for the formset children. This is mostly used internally, and rarely needs to be used by external projects. When using the factory methods (:func:'polymorphic_inlineformset_factory'), this feature is called already for you.

class polymorphic.formsets.**BasePolymorphicModelFormSet**(**args*, ***kwargs*)

Bases: `django.forms.models.BaseModelFormSet`

A formset that can produce different forms depending on the object type.

Note that the 'add' feature is therefore more complex, as all variations need ot be exposed somewhere.

When switching existing formsets to the polymorphic formset, note that the ID field will no longer be named ''model_ptr'', but just appear as ''id''.

class polymorphic.formsets.**BasePolymorphicInlineFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)

Bases: `django.forms.models.BaseInlineFormSet`, `polymorphic.formsets.models.BasePolymorphicModelFormSet`

Polymorphic formset variation for inline formsets

class polymorphic.formsets.**BaseGenericPolymorphicInlineFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=None*, *prefix=None*, *queryset=None*, ***kwargs*)

Bases: `django.contrib.contenttypes.forms.BaseGenericInlineFormSet`, `polymorphic.formsets.models.BasePolymorphicModelFormSet`

Polymorphic formset variation for inline generic formsets

3.7.5 polymorphic.managers

The manager class for use in the models.

The `PolymorphicManager` class

class polymorphic.managers.**PolymorphicManager**

Bases: `django.db.models.manager.Manager`

Manager for PolymorphicModel

Usually not explicitly needed, except if a custom manager or a custom queryset class is to be used.

queryset_class

alias of *PolymorphicQuerySet*

The `PolymorphicQuerySet` class

class polymorphic.managers.**PolymorphicQuerySet**(**args*, ***kwargs*)

Bases: `django.db.models.query.QuerySet`

QuerySet for PolymorphicModel

Contains the core functionality for PolymorphicModel

Usually not explicitly needed, except if a custom queryset class is to be used.

aggregate(**args*, ***kwargs*)

translate the polymorphic field paths in the kwargs, then call vanilla aggregate. We need no polymorphic object retrieval for aggregate => switch it off.

annotate(**args*, ***kwargs*)

translate the polymorphic field paths in the kwargs, then call vanilla annotate. _get_real_instances will do the rest of the job after executing the query.

defer(**fields*)
Translate the field paths in the args, then call vanilla defer.

Also retain a copy of the original fields passed, which we'll need when we're retrieving the real instance (since we'll need to translate them again, as the model will have changed).

get_real_instances(*base_result_objects=None*)
Cast a list of objects to their actual classes.

This does roughly the same as:

```
return [ o.get_real_instance() for o in base_result_objects ]
```

but more efficiently.

Return type *PolymorphicQuerySet*

instance_of(**args*)
Filter the queryset to only include the classes in args (and their subclasses).

non_polymorphic()
switch off polymorphic behaviour for this query. When the queryset is evaluated, only objects of the type of the base class used for this query are returned.

not_instance_of(**args*)
Filter the queryset to exclude the classes in args (and their subclasses).

only(**fields*)
Translate the field paths in the args, then call vanilla only.

Also retain a copy of the original fields passed, which we'll need when we're retrieving the real instance (since we'll need to translate them again, as the model will have changed).

order_by(**field_names*)
translate the field paths in the args, then call vanilla order_by.

3.7.6 polymorphic.models

Seamless Polymorphic Inheritance for Django Models

class polymorphic.models.**PolymorphicModel**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Abstract base class that provides polymorphic behaviour for any model directly or indirectly derived from it.

PolymorphicModel declares one field for internal use (*polymorphic_ctype*) and provides a polymorphic manager as the default manager (and as 'objects').

Parameters **polymorphic_ctype_id** (ForeignKey to ContentType) – Polymorphic ctype

get_real_instance()
Upcast an object to it's actual type.

If a non-polymorphic manager (like base_objects) has been used to retrieve objects, then the complete object with it's real class/type and all fields may be retrieved with this method.

Note: Each method call executes one db query (if necessary). Use the *get_real_instances()* to upcast a complete list in a single efficient query.

---

get_real_instance_class()
Return the actual model type of the object.

If a non-polymorphic manager (like base_objects) has been used to retrieve objects, then the real class/type of these objects may be determined using this method.

pre_save_polymorphic(*using='default'*)
Make sure the `polymorphic_ctype` value is correctly set on this model.

save(**args*, ***kwargs*)
Calls *`pre_save_polymorphic()`* and saves the model.

polymorphic_ctype
The model field that stores the `ContentType` reference to the actual class.

3.7.7 polymorphic.templatetags.polymorphic_admin_tags

Template tags for polymorphic

The `polymorphic_formset_tags` Library

New in version 1.1.

To render formsets in the frontend, the `polymorphic_tags` provides extra filters to implement HTML rendering of polymorphic formsets.

The following filters are provided;

- `{{ formset|as_script_options }}` render the `data-options` for a JavaScript formset library.
- `{{ formset|include_empty_form }}` provide the placeholder form for an add button.
- `{{ form|as_form_type }}` return the model name that the form instance uses.
- `{{ model|as_model_name }}` performs the same, for a model class or instance.

```
{% load i18n polymorphic_formset_tags %}

<div class="inline-group" id="{{ formset.prefix }}-group" data-options="{{ formset|as_
↪script_options }}">
    {% block add_button %}
        {% if formset.show_add_button|default_if_none:'1' %}
            {% if formset.empty_forms %}
                {# django-polymorphic formset (e.g. PolymorphicInlineFormSetView) #}
                <div class="btn-group" role="group">
                  {% for model in formset.child_forms %}
                      <a type="button" data-type="{{ model|as_model_name }}" class=
↪"js-add-form btn btn-default">{% glyphicon 'plus' %} {{ model|as_verbose_name }}</a>
                  {% endfor %}
                </div>
            {% else %}
                <a class="btn btn-default js-add-form">{% trans "Add" %}</a>
            {% endif %}
        {% endif %}
    {% endblock %}

    {{ formset.management_form }}

    {% for form in formset|include_empty_form %}
```

```
      {% block formset_form_wrapper %}
        <div id="{{ form.prefix }}" data-inline-type="{{ form|as_form_type|lower }}"␣
↪class="inline-related{% if '__prefix__' in form.prefix %} empty-form{% endif %}">
            {{ form.non_field_errors }}

            {# Add the 'pk' field that is not mentioned in crispy #}
            {% for field in form.hidden_fields %}
              {{ field }}
            {% endfor %}

            {% block formset_form %}
                {% crispy form %}
            {% endblock %}
        </div>
      {% endblock %}
    {% endfor %}
</div>
```

The `polymorphic_admin_tags` Library

The `{% breadcrumb_scope ... %}` tag makes sure the `{{ opts }}` and `{{ app_label }}` values are temporary based on the provided `{{ base_opts }}`. This allows fixing the breadcrumb in admin templates:

```
{% extends "admin/change_form.html" %}
{% load polymorphic_admin_tags %}

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

3.7.8 polymorphic.utils

`polymorphic.utils.`**`get_base_polymorphic_model`**(*ChildModel*, *allow_abstract=False*)

First the first concrete model in the inheritance chain that inherited from the PolymorphicModel.

`polymorphic.utils.`**`reset_polymorphic_ctype`**(**models*, ***filters*)

Set the polymorphic content-type ID field to the proper model Sort the `*models` from base class to descending class, to make sure the content types are properly assigned.

Add `preserve_existing=True` to skip models which already have a polymorphic content type.

`polymorphic.utils.`**`sort_by_subclass`**(**classes*)

Sort a series of models by their inheritance order.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

E

F

G

Q

R

S

T