django-parler Documentation

Release 1.9.2

Diederik van der Boor and contributors

Aug 27, 2018

Contents

"Easily translate "cheese omelet" into "omelette du fromage".

django-parler provides Django model translations without nasty hacks.

Features:

- Nice admin integration.
- Access translated attributes like regular attributes.
- Automatic fallback to the other default languages.
- Separate table for translated fields, compatible with django-hvad.
- Plays nice with others, compatible with django-polymorphic, django-mptt and such:
- No ORM query hacks.
- Easy to combine with custom Manager or QuerySet classes.
- Easy to construct the translations model manually when needed.

CHAPTER 1

Getting started

1.1 Quick start guide

1.1.1 Installing django-parler

The package can be installed using:

```
pip install django-parler
```

Add the following settings:

```
INSTALLED_APPS += (
    'parler',
)
```

1.1.2 A brief overview

Creating models

Using the *TranslatedFields* wrapper, model fields can be marked as translatable:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFields

class MyModel(TranslatableModel):
    translations = TranslatedFields(
        title = models.CharField(_("Title"), max_length=200)
    )

    def __unicode__(self):
        return self.title
```

Accessing fields

Translatable fields can be used like regular fields:

```
>>> object = MyModel.objects.all()[0]
>>> object.get_current_language()
'en'
>>> object.title
u'cheese omelet'

>>> object.set_current_language('fr')        # Only switches
>>> object.title = "omelette du fromage"     # Translation is created on demand.
>>> object.save()
```

Internally, django-parler stores the translated fields in a separate model, with one row per language.

Filtering translations

To query translated fields, use the *translated()* method:

```
MyObject.objects.translated(title='cheese omelet')
```

To access objects in both the current and the configured fallback languages, use:

```
MyObject.objects.active_translations(title='cheese omelet')
```

This returns objects in the languages which are considered "active", which are:

- The current language
- The fallback languages when `hide_untranslated=False` in the *PARLER_LANGUAGES* setting.

Note: Due to *ORM restrictions* the query should be performed in a single *translated()* or *active_translations()* call.

The *active_translations()* method typically needs to include a `distinct()` call to avoid duplicate results of the same object.

Changing the language

The queryset can be instructed to return objects in a specific language:

```
>>> objects = MyModel.objects.language('fr').all()
>>> objects[0].title
u'omelette du fromage'
```

This only sets the language of the object. By default, the current Django language is used.

Use `get_current_language()` and `set_current_language()` to change the language on individual objects. There is a context manager to do this temporary:

```
from parler.utils.context import switch_language

with switch_language(model, 'fr'):
    print model.title
```

And a function to query just a specific field:

```
model.safe_translation_getter('title', language_code='fr')
```

1.1.3 Configuration

By default, the fallback languages are the same as: `[LANGUAGE_CODE]`. The fallback language can be changed in the settings:

```
PARLER_DEFAULT_LANGUAGE_CODE = 'en'
```

Optionally, the admin tabs can be configured too:

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en',},
        {'code': 'en-us',},
        {'code': 'it',},
        {'code': 'nl',},
    ),
    'default': {
        'fallbacks': ['en'],          # defaults to PARLER_DEFAULT_LANGUAGE_CODE
        'hide_untranslated': False,   # the default; let .active_translations()␣
↪return fallbacks too.
    }
}
```

Replace `None` with the `SITE_ID` when you run a multi-site project with the sites framework. Each `SITE_ID` can be added as additional entry in the dictionary.

1.2 Configuration options

1.2.1 PARLER_DEFAULT_LANGUAGE_CODE

The language code for the fallback language. This language is used when a translation for the currently selected language does not exist.

By default, it's the same as `LANGUAGE_CODE`.

This value is used as input for `PARLER_LANGUAGES['default']['fallback']`.

1.2.2 PARLER_LANGUAGES

The configuration of language defaults. This is used to determine the languages in the ORM and admin.

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en',},
        {'code': 'en-us',},
        {'code': 'it',},
        {'code': 'nl',},
    ),
    'default': {
```

(continues on next page)

(continued from previous page)

```
        'fallbacks': ['en'],          # defaults to PARLER_DEFAULT_LANGUAGE_CODE
        'hide_untranslated': False,   # the default; let .active_translations()␣
↪return fallbacks too.
    }
}
```

The values in the `default` section are applied to all entries in the dictionary, filling any missing values.

The following entries are available:

`code` The language code for the entry.

`fallbacks` The fallback languages for the entry

> Changed in version 1.5: In the previous versions, this field was called `fallback` and pointed to a single language. The old setting name is still supported, but it's recommended you upgrade your settings.

`hide_untranslated` Whether untranslated objects should be returned by *`active_translations()`*.

- When `True`, only the current language is returned, and no fallback language is used.
- When `False`, objects having either a translation or fallback are returned.

The default is `False`.

Multi-site support

When using the sites framework (`django.contrib.sites`) and the `SITE_ID` setting, the dict can contain entries for every site ID. The special `None` key is no longer used:

```
PARLER_LANGUAGES = {
    # Global site
    1: (
        {'code': 'en',},
        {'code': 'en-us',},
        {'code': 'it',},
        {'code': 'nl',},
    ),
    # US site
    2: (
        {'code': 'en-us',},
        {'code': 'en',},
    ),
    # IT site
    3: (
        {'code': 'it',},
        {'code': 'en',},
    ),
    # NL site
    3: (
        {'code': 'nl',},
        {'code': 'en',},
    ),
    'default': {
        'fallbacks': ['en'],          # defaults to PARLER_DEFAULT_LANGUAGE_CODE
        'hide_untranslated': False,   # the default; let .active_translations()␣
↪return fallbacks too.
    }
}
```

In this example, each language variant only display 2 tabs in the admin, while the global site has an overview of all languages.

1.2.3 PARLER_ENABLE_CACHING

```
PARLER_ENABLE_CACHING = True
```

This setting is strictly for experts or for troubleshooting situations, where disabling caching can be beneficial.

1.2.4 PARLER_SHOW_EXCLUDED_LANGUAGE_TABS

```
PARLER_SHOW_EXCLUDED_LANGUAGE_TABS = False
```

By default, the admin tabs are limited to the language codes found in `LANGUAGES`. If the models have other translations, they can be displayed by setting this value to `True`.

1.3 Template tags

All translated fields can be read like normal fields, just using like:

```
{{ object.fieldname }}
```

When a translation is not available for the field, an empty string (or `TEMPLATE_STRING_IF_INVALID`) will be outputted. The Django template system safely ignores the *`TranslationDoesNotExist`* exception that would normally be emitted in code; that's because that exception inherits from `AttributeError`.

For other situations, you may need to use the template tags, e.g.:

- Getting a translated URL of the current page, or any other object.
- Switching the object language, e.g. to display fields in a different language.
- Fetching translated fields in a thread-safe way (for shared objects).

To use the template loads, add this to the top of the template:

```
{% load parler_tags %}
```

1.3.1 Getting the translated URL

The `get_translated_url` tag can be used to get the proper URL for this page in a different language. If the URL could not be generated, an empty string is returned instead.

This algorithm performs a "best effort" approach to give a proper URL. When this fails, add the *`ViewUrlMixin`* to your view to contruct the proper URL instead.

Example, to build a language menu:

```
{% load i18n parler_tags %}

<ul>
    {% for lang_code, title in LANGUAGES %}
        {% get_language_info for lang_code as lang %}
```

(continues on next page)

(continued from previous page)

```
        {% get_translated_url lang_code as tr_url %}
        {% if tr_url %}<li{% if lang_code == LANGUAGE_CODE %} class="is-selected"{%␣
↪endif %}><a href="{{ tr_url }}" hreflang="{{ lang_code }}">{{ lang.name_
↪local|capfirst }}</a></li>{% endif %}
    {% endfor %}
</ul>
```

To inform search engines about the translated pages:

```
{% load i18n parler_tags %}

{% for lang_code, title in LANGUAGES %}
    {% get_translated_url lang_code as tr_url %}
    {% if tr_url %}<link rel="alternate" hreflang="{{ lang_code }}" href="{{ tr_url }}
↪" />{% endif %}
{% endfor %}
```

Note: Using this tag is not thread-safe if the object is shared between threads. It temporary changes the current language of the object.

1.3.2 Changing the object language

To switch an object language, use:

```
{% objectlanguage object "en" %}
  {{ object.title }}
{% endobjectlanguage %}
```

A *TranslatableModel* is not affected by the `{% language .. %}` tag as it maintains it's own state. Using this tag temporary switches the object state.

Note: Using this tag is not thread-safe if the object is shared between threads. It temporary changes the current language of the object.

1.3.3 Thread safety notes

Using the `{% get_translated_url %}` or `{% objectlanguage %}` tags is not thread-safe if the object is shared between threads. It temporary changes the current language of the view object. Thread-safety is rarely an issue in templates, when all objects are fetched from the database in the view.

One example where it may happen, is when you have objects cached in global variables. For example, attaching objects to the `Site` model causes this. A shared object is returned when these objects are accessed using `Site.objects.get_current().my_object`. That's because the sites framework keeps a global cache of all `Site` objects, and the `my_object` relationship is also cached by the ORM. Hence, the object is shared between all requests.

In case an object is shared between threads, a safe way to access the translated field is by using the template filter `get_translated_field` or your own variation of it:

```
{{ object|get_translated_field:'name' }}
```

This avoids changing the `object` language with a `set_current_language()` call. Instead, it directly reads the translated field using `safe_translation_getter()`. The field is fetched in the current Django template, and follows the project language settings (whether to use fallbacks, and `any_language` setting).

1.4 Performance guidelines

The translations of each model is stored in a separate table. In some cases, this may cause in N-query issue. *django-parler* offers two ways to handle the performance of the dabase.

1.4.1 Caching

All translated contents is cached by default. Hence, when an object is read again, no query is performed. This works out of the box when the project uses a proper caching:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'KEY_PREFIX': 'mysite.production',  # Change this
        'LOCATION': '127.0.0.1:11211',
        'TIMEOUT': 24*3600
    },
}
```

You have to make sure your project has the proper backend support available:

```
pip install python-memcached
```

Now, the translation table only has to be read once per day.

1.4.2 Query prefetching

By using `prefetch_related()`, all translations can be fetched in a single query:

```
object_list = MyModel.objects.prefetch_related('translations')
for obj in object_list:
    print obj.title  # reads translated title from the prefetched queryset
```

Note that the prefetch reads the information of all languages, not just the currently active language.

When you display translated objects in a form, e.g. a select list, you can prefetch the queryset too:

```
class MyModelAdminForm(TranslatableModelForm):
    def __init__(self, *args, **kwargs):
        super(MyModelAdminForm, self).__init__(*args, **kwargs)
        self.fields['some_field'].queryset = self.fields['some_field'].queryset.
↪prefetch_related('translations')
```

CHAPTER 2

In depth topics

2.1 Advanced usage patterns

2.1.1 Translations without fallback languages

When a translation is missing, the fallback languages are used. However, when an object has no fallback languages, this still fails.

There are a few solutions to this problem:

1. Declare the translated attribute explicitly with `any_language=True`:

   ```
   from parler.models import TranslatableModel
   from parler.fields import TranslatedField

   class MyModel(TranslatableModel):
       title = TranslatedField(any_language=True)
   ```

   Now, the title will try to fetch one of the existing languages from the database.

2. Use `safe_translation_getter()` on attributes which don't have an `any_language=True` setting. For example:

   ```
   model.safe_translation_getter("fieldname", any_language=True)
   ```

3. Catch the *`TranslationDoesNotExist`* exception. For example:

   ```
   try:
       return object.title
   except TranslationDoesNotExist:
       return ''
   ```

   Because this exception inherits from `AttributeError`, templates already display empty values by default.

4. Avoid fetching untranslated objects using queryset methods. For example:

```
queryset.active_translations()
```

Which is almost identical to:

```
codes = get_active_language_choices()
queryset.filter(translations__language_code__in=codes).distinct()
```

Note that the same ORM restrictions apply here.

2.1.2 Using translated slugs in views

To handle translatable slugs in the `DetailView`, the *`TranslatableSlugMixin`* can be used to make this work smoothly. For example:

```
class ArticleDetailView(TranslatableSlugMixin, DetailView):
    model = Article
    template_name = 'article/details.html'
```

The *`TranslatableSlugMixin`* makes sure that:

- The object is fetched in the proper translation.
- The slug field is read from the translation model, instead of the shared model.
- Fallback languages are handled.
- Objects are not accidentally displayed in their fallback slugs, but redirect to the translated slug.

2.1.3 Making existing fields translatable

The following guide explains how to make existing fields translatable, and migrate the data from the old fields to translated fields.

django-parler stores translated fields in a separate model, so it can store multiple versions (translations) of the same field. To make existing fields translatable, 3 migration steps are needed:

1. Create the translation table, keep the existing columns
2. Copy the data from the original table to the translation table.
3. Remove the fields from the original model.

The following sections explain this in detail:

Step 1: Create the translation table

Say we have the following model:

```
class MyModel(models.Model):
    name = models.CharField(max_length=123)
```

First create the translatable fields:

```
class MyModel(TranslatableModel):
    name = models.CharField(max_length=123)

    translations = TranslatedFields(
          name=models.CharField(max_length=123),
    )
```

Now create the migration:

- For Django 1.7, use: `manage.py makemigrations myapp "add_translation_model"`
- For Django 1.8 and above, use: `manage.py makemigrations myapp --name "add_translation_model"`

Step 2: Copy the data

Within the data migration, copy the existing data:

Create an empty migration:

```
manage.py makemigrations --empty myapp --name "migrate_translatable_fields"
```

And use it to move the data:

```
from django.db import migrations
from django.conf import settings
from django.core.exceptions import ObjectDoesNotExist


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

    for object in MyModel.objects.all():
        MyModelTranslation.objects.create(
            master_id=object.pk,
            language_code=settings.LANGUAGE_CODE,
            name=object.name
        )

def backwards_func(apps, schema_editor):
    MyModel = apps.get_model('myapp', 'MyModel')
    MyModelTranslation = apps.get_model('myapp', 'MyModelTranslation')

    for object in MyModel.objects.all():
        translation = _get_translation(object, MyModelTranslation)
        object.name = translation.name
        object.save()    # Note this only calls Model.save()

def _get_translation(object, MyModelTranslation):
    translations = MyModelTranslation.objects.filter(master_id=object.pk)
    try:
        # Try default translation
        return translations.get(language_code=settings.LANGUAGE_CODE)
    except ObjectDoesNotExist:
        try:
            # Try default language
```

(continues on next page)

(continued from previous page)

```
            return translations.get(language_code=settings.PARLER_DEFAULT_LANGUAGE_
↪CODE)
        except ObjectDoesNotExist:
            # Maybe the object was translated only in a specific language?
            # Hope there is a single translation
            return translations.get()


class Migration(migrations.Migration):

    dependencies = [
        ('yourappname', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(forwards_func, backwards_func),
    ]
```

Note: Be careful which language is used to migrate the existing data. In this example, the `backwards_func()` logic is extremely defensive not to loose translated data.

Step 3: Remove the old fields

Remove the old field from the original model. The example model now looks like:

```
class MyModel(TranslatableModel):
    translations = TranslatedFields(
        name=models.CharField(max_length=123),
    )
```

Create the database migration, it will simply remove the original field.

- For Django 1.7, use: `manage.py makemigrations myapp "remove_untranslated_fields"`
- For Django 1.8 and above, use: `manage.py makemigrations myapp --name "remove_untranslated_fields"`

Updating code

The project code should be updated. For example:

- Replace `filter(field_name)` with `.translated(field_name)` or `filter(translations__field_name)`.
- Make sure there is one filter on the translated fields, see *Using multiple filter() calls*.
- Update the `ordering` and `order_by()` code. See *The ordering meta field*.
- Update the admin `search_fields` and `prepopulated_fields`. See *Using search_fields in the admin*.

Deployment

To have a smooth deployment, it's recommended to only run the first 2 migrations - which create columns and move the data. Removing the old fields should be done after reloading the WSGI instance.

2.1.4 Adding translated fields to an existing model

Create a proxy class:

```
from django.contrib.sites.models import Site
from parler.models import TranslatableModel, TranslatedFields


class TranslatableSite(TranslatableModel, Site):
    class Meta:
        proxy = True

    translations = TranslatedFields()
```

And update the admin:

```
from django.contrib.sites.admin import SiteAdmin
from django.contrib.sites.models import Site
from parler.admin import TranslatableAdmin, TranslatableStackedInline


class NewSiteAdmin(TranslatableAdmin, SiteAdmin):
    pass

admin.site.unregister(Site)
admin.site.register(TranslatableSite, NewSiteAdmin)
```

Overwriting existing untranslated fields

Note that it is not possible to add translations in the proxy class with the same name as fields in the parent model. This will not show up as an error yet, but it will fail when the objects are fetched from the database. Instead, opt for reading *Making existing fields translatable*.

2.1.5 Integration with django-mptt

When you have to combine *TranslatableModel* with MPTTModel you have to make sure the model managers of both classes are combined too.

This can be done by extending the Manager and QuerySet class.

Note: This example is written for django-mptt >= 0.7.0, which also requires combining the queryset classes.

For a working example, see django-categories-i18n.

Combining TranslatableModel with MPTTModel

Say we have a base Category model that needs to be translatable:

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible, force_text
from parler.models import TranslatableModel, TranslatedFields
from parler.managers import TranslatableManager
```

(continues on next page)

(continued from previous page)

```
from mptt.models import MPTTModel
from .managers import CategoryManager


@python_2_unicode_compatible
class Category(MPTTModel, TranslatableModel):
    # The shared base model. Either place translated fields here,
    # or place them at the subclasses (see note below).
    parent = models.ForeignKey('self', related_name='children')

    translations = TranslatedFields(
        name=models.CharField(blank=False, default='', max_length=128),
        slug=models.SlugField(blank=False, default='', max_length=128)
    )

    objects = CategoryManager()

    def __str__(self):
        return self.safe_translation_getter('name', any_language=True)
```

Combining managers

The managers can be combined by inheriting them. Unfortunately, django-mptt 0.7 overrides the `get_queryset()` method, so it needs to be redefined:

```
import django
from parler.managers import TranslatableManager, TranslatableQuerySet
from mptt.managers import TreeManager
from mptt.querysets import TreeQuerySet  # new as of mptt 0.7


class CategoryQuerySet(TranslatableQuerySet, TreeQuerySet):
    pass

    # Optional: make sure the Django 1.7 way of creating managers works.
    def as_manager(cls):
        manager = CategoryManager.from_queryset(cls)()
        manager._built_with_as_manager = True
        return manager
    as_manager.queryset_only = True
    as_manager = classmethod(as_manager)


class CategoryManager(TreeManager, TranslatableManager):
    queryset_class = CategoryQuerySet

    def get_queryset(self):
        # This is the safest way to combine both get_queryset() calls
        # supporting all Django versions and MPTT 0.7.x versions
        return self.queryset_class(self.model, using=self._db).order_by(self.tree_id_
↪attr, self.left_attr)

    if django.VERSION < (1,6):
        get_query_set = get_queryset
```

Assign the manager to the model `objects` attribute.

Implementing the admin

By merging the base classes, the admin interface supports translatable MPTT models:

```
from django.contrib import admin
from parler.admin import TranslatableAdmin, TranslatableModelForm
from mptt.admin import MPTTModelAdmin
from mptt.forms import MPTTAdminForm
from .models import Category


class CategoryAdminForm(MPTTAdminForm, TranslatableModelForm):
    pass


class CategoryAdmin(TranslatableAdmin, MPTTModelAdmin):
    form = CategoryAdminForm

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('title',)}  # needed for translated fields


admin.site.register(Category, CategoryAdmin)
```

2.1.6 Integration with django-polymorphic

When you have to combine *TranslatableModel* with PolymorphicModel you have to make sure the model managers of both classes are combined too.

This can be done by either overwriting default_manager or by extending the Manager and QuerySet class.

Combining TranslatableModel with PolymorphicModel

Say we have a base Product with two concrete products, a Book with two translatable fields name and slug, and a Pen with one translatable field identifier. Then the following pattern works for a polymorphic Django model:

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible, force_text
from parler.models import TranslatableModel, TranslatedFields
from parler.managers import TranslatableManager
from polymorphic import PolymorphicModel
from .managers import BookManager


class Product(PolymorphicModel):
    # The shared base model. Either place translated fields here,
    # or place them at the subclasses (see note below).
    code = models.CharField(blank=False, default='', max_length=16)
    price = models.DecimalField(max_digits=10, decimal_places=2, default=0.00)


@python_2_unicode_compatible
class Book(Product, TranslatableModel):
    # Solution 1: use a custom manager that combines both.
    objects = BookManager()
```

(continues on next page)

(continued from previous page)

```
    translations = TranslatedFields(
        name=models.CharField(blank=False, default='', max_length=128),
        slug=models.SlugField(blank=False, default='', max_length=128)
    )

    def __str__(self):
        return force_text(self.code)


@python_2_unicode_compatible
class Pen(Product, TranslatableModel):
    # Solution 2: override the default manager.
    default_manager = TranslatableManager()

    translations = TranslatedFields(
        identifier=models.CharField(blank=False, default='', max_length=255)
    )

    def __str__(self):
        return force_text(self.identifier)
```

The only precaution one must take, is to override the default manager in each of the classes containing translatable fields. This is shown in the example above.

As of django-parler 1.2 it's possible to have translations on both the base and derived models. Make sure that the field name (in this case `translations`) differs between both models, as that name is used as `related_name` for the translated fields model

Combining managers

The managers can be combined by inheriting them, and specifying the *queryset_class* attribute with both *django-parler* and django-polymorphic use.

```
from parler.managers import TranslatableManager, TranslatableQuerySet
from polymorphic import PolymorphicManager
from polymorphic.query import PolymorphicQuerySet


class BookQuerySet(TranslatableQuerySet, PolymorphicQuerySet):
    pass

class BookManager(PolymorphicManager, TranslatableManager):
    queryset_class = BookQuerySet
```

Assign the manager to the model `objects` attribute.

Implementing the admin

It is perfectly possible to 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.

This admin interface adds translatable fields to a polymorphic model:

```
from django.contrib import admin
from parler.admin import TranslatableAdmin, TranslatableModelForm
from polymorphic.admin import PolymorphicParentModelAdmin, PolymorphicChildModelAdmin
from .models import BaseProduct, Book, Pen


class BookAdmin(TranslatableAdmin, PolymorphicChildModelAdmin):
    base_form = TranslatableModelForm
    base_model = BaseProduct
    base_fields = ('code', 'price', 'name', 'slug')

class PenAdmin(TranslatableAdmin, PolymorphicChildModelAdmin):
    base_form = TranslatableModelForm
    base_model = BaseProduct
    base_fields = ('code', 'price', 'identifier',)

class BaseProductAdmin(PolymorphicParentModelAdmin):
    base_model = BaseProduct
    child_models = ((Book, BookAdmin), (Pen, PenAdmin),)
    list_display = ('code', 'price',)

admin.site.register(BaseProduct, BaseProductAdmin)
```

2.1.7 Integration with django-guardian

Combining `TranslatableAdmin` with `GuardedModelAdmin`

To combine the `TranslatableAdmin` with the `GuardedModelAdmin` from django-guardian there are a few things to notice.

Depending on the order of inheritance, either the parler language tabs or guardian "Object permissions" button may not be visible anymore.

To fix this you'll have to make sure both template parts are included in the page.

Both classes override the `change_form_template` value:

- `GuardedModelAdmin` sets it to `admin/guardian/model/change_form.html` explicitly.
- `TranslatableAdmin` sets it to `admin/parler/change_form.html`, but it inherits the original template that the admin would have auto-selected otherwise.

Using `TranslatableAdmin` as first class

When the `TranslatableAdmin` is the first inherited class:

```
class ProjectAdmin(TranslatableAdmin, GuardedModelAdmin):
    pass
```

You can create a template such as `myapp/project/change_form.html` which inherits the guardian template:

```
{% extends "admin/guardian/model/change_form.html" %}
```

Now, *django-parler* will load this template in `admin/parler/change_form.html`, so both the guardian and parler content is visible.

Using `GuardedModelAdmin` as first class

When the `GuardedModelAdmin` is the first inherited class:

```
class ProjectAdmin(TranslatableAdmin, GuardedModelAdmin):
    change_form_template = 'myapp/project/change_form.html'
```

The `change_form_template` needs to be set manually. It can either be set to `admin/parler/change_form.html`, or use a custom template that includes both bits:

```
{% extends "admin/guardian/model/change_form.html" %}

{# restore django-parler tabs #}
{% block field_sets %}
{% include "admin/parler/language_tabs.html" %}
{{ block.super }}
{% endblock %}
```

2.1.8 Integration with django-rest-framework

To integrate the translated fields in django-rest-framework, the django-parler-rest module provides serializer fields. These fields can be used to integrate translations into the REST output.

Example code

The following Country model will be exposed:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFields

class Country(TranslatableModel):
    code = models.CharField(_("Country code"), max_length=2, unique=True, primary_
↪key=True, db_index=True)

    translations = TranslatedFields(
        name = models.CharField(_("Name"), max_length=200, blank=True)
    )

    def __unicode__(self):
        self.name

    class Meta:
        verbose_name = _("Country")
        verbose_name_plural = _("Countries")
```

The following code is used in the serializer:

```
from parler_rest.serializers import TranslatableModelSerializer
from parler_rest.fields import TranslatedFieldsField
from myapp.models import Country

class CountrySerializer(TranslatableModelSerializer):
    translations = TranslatedFieldsField(shared_model=Country)
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = Country
        fields = ('code', 'translations')
```

2.1.9 Multi-site support

When using the sites framework (`django.contrib.sites`) and the `SITE_ID` setting, the dict can contain entries for every site ID. See the *configuration* for more details.

2.1.10 Disabling caching

If desired, caching of translated fields can be disabled by adding *PARLER_ENABLE_CACHING = False* to the settings.

2.1.11 Constructing the translations model manually

It's also possible to create the translated fields model manually:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFieldsModel
from parler.fields import TranslatedField


class MyModel(TranslatableModel):
    title = TranslatedField()  # Optional, explicitly mention the field

    class Meta:
        verbose_name = _("MyModel")

    def __unicode__(self):
        return self.title


class MyModelTranslation(TranslatedFieldsModel):
    master = models.ForeignKey(MyModel, related_name='translations', null=True)
    title = models.CharField(_("Title"), max_length=200)

    class Meta:
        unique_together = ('language_code', 'master')
        verbose_name = _("MyModel translation")
```

This has the same effect, but also allows to to override the `save()` method, or add new methods yourself.

2.1.12 Customizing language settings

If needed, projects can "fork" the parler language settings. This is rarely needed. Example:

```
from django.conf import settings
from parler import appsettings as parler_appsettings
from parler.utils import normalize_language_code, is_supported_django_language
from parler.utils.conf import add_default_language_settings
```

(continues on next page)

(continued from previous page)

```
MYCMS_DEFAULT_LANGUAGE_CODE = getattr(settings, 'MYCMS_DEFAULT_LANGUAGE_CODE', FLUENT_
↪DEFAULT_LANGUAGE_CODE)
MYCMS_LANGUAGES = getattr(settings, 'MYCMS_LANGUAGES', parler_appsettings.PARLER_
↪LANGUAGES)

MYCMS_DEFAULT_LANGUAGE_CODE = normalize_language_code(MYCMS_DEFAULT_LANGUAGE_CODE)

MYCMS_LANGUAGES = add_default_language_settings(
    MYCMS_LANGUAGES, 'MYCMS_LANGUAGES',
    hide_untranslated=False,
    hide_untranslated_menu_items=False,
    code=MYCMS_DEFAULT_LANGUAGE_CODE,
    fallbacks=[MYCMS_DEFAULT_LANGUAGE_CODE]
)
```

Instead of using the functions from *parler.utils* (such as *get_active_language_choices()*) the project can access the language settings using:

```
MYCMS_LANGUAGES.get_language()
MYCMS_LANGUAGES.get_active_choices()
MYCMS_LANGUAGES.get_fallback_languages()
MYCMS_LANGUAGES.get_default_language()
MYCMS_LANGUAGES.get_first_language()
```

These methods are added by the *add_default_language_settings()* function. See the *LanguagesSetting* class for details.

2.2 Django compatibility

This package has been tested with:

- Django versions 1.7 up to 1.11
- Python versions 2.7 up to 3.6

2.2.1 Using multiple `filter()` calls

Since translated fields live in a separate model, they can be filtered like any normal relation:

```
object = MyObject.objects.filter(translations__title='cheese omelet')

translation1 = myobject.translations.all()[0]
```

However, if you have to query a language or translated attribute, this should happen in a single query. That can either be a single `filter()`, *translated()* or *active_translations()*) call:

```
from parler.utils import get_active_language_choices

MyObject.objects.filter(
    translations__language_code__in=get_active_language_choices(),
    translations__slug='omelette'
)
```

Queries on translated fields, even just `.translated()` spans a relationship. Hence, they can't be combined with other filters on translated fields, as that causes double joins on the translations table. See the ORM documentation for more details.

2.2.2 The `ordering` meta field

It's not possible to order on translated fields by default. Django won't allow the following:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFields

class MyModel(TranslatableModel):
    translations = TranslatedFields(
        title = models.CharField(max_length=100),
    )

    class Meta:
        ordering = ('title',)  # NOT ALLOWED

    def __unicode__(self):
        return self.title
```

You can however, perform ordering within the queryset:

```
MyModel.objects.translated('en').order_by('translations__title')
```

You can also use the provided classes to perform the sorting within Python code.

- For the admin `list_filter` use: `SortedRelatedFieldListFilter`
- For forms widgets use: *`SortedSelect`*, *`SortedSelectMultiple`*, *`SortedCheckboxSelectMultiple`*

2.2.3 Using `search_fields` in the admin

When translated fields are included in the `search_fields`, they should be includes with their full ORM path. For example:

```
from parler.admin import TranslatableAdmin

class MyModelAdmin(TranslatableAdmin):
    search_fields = ('translations__title',)
```

2.2.4 Using `prepopulated_fields` in the admin

Using `prepopulated_fields` doesn't work yet, as the admin will complain that the field does not exist. Use `get_prepopulated_fields()` as workaround:

```
from parler.admin import TranslatableAdmin

class MyModelAdmin(TranslatableAdmin):

    def get_prepopulated_fields(self, request, obj=None):
        # can't use `prepopulated_fields = ..` because it breaks the admin validation
```

(continues on next page)

(continued from previous page)

```
        # for translated fields. This is the official django-parler workaround.
        return {
            'slug': ('title',)
        }
```

2.3 Background

2.3.1 A brief history

This package is inspired by django-hvad. When attempting to integrate multilingual support into django-fluent-pages using django-hvad this turned out to be really hard. The sad truth is that while django-hvad has a nice admin interface, table layout and model API, it also overrides much of the default behavior of querysets and model metaclasses. This prevents combining django-hvad with django-polymorphic or django-mptt for example.

When investigating other multilingual packages, they either appeared to be outdated, store translations in the same table (too inflexible for us) or only provided a model API. Hence, there was a need for a new solution, using a simple, crude but effective API.

To start multilingual support in our django-fluent-pages package, it was coded directly into the package itself. A future django-hvad transition was kept in mind. Instead of doing metaclass operations, the "shared model" just proxied all attributes to the translated model (all manually constructed). Queries just had to be performed using `.filter(translations__title=..)`. This proved to be a sane solution and quickly it turned out that this code deserved a separate package, and some other modules needed it too.

This package is an attempt to combine the best of both worlds; the API simplicity of django-hvad with the crude, but effective solution of proxying translated attributes.

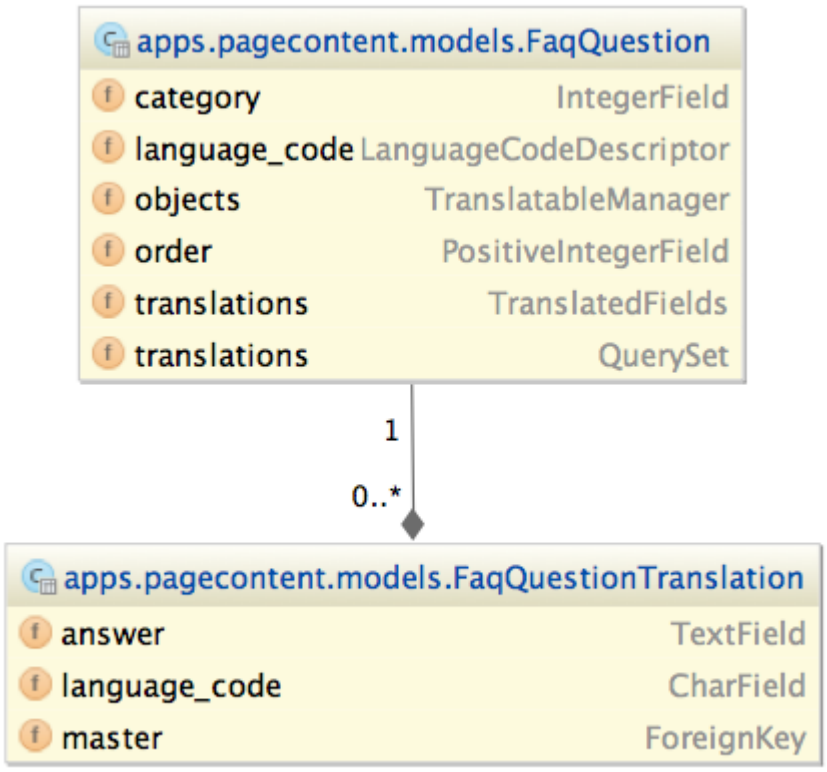
Added on top of that, the API-sugar is provided, similar to what django-hvad has. It's possible to create the translations model manually, or let it be created dynamically when using the *`TranslatedFields`* field. This is to make your life easier - without loosing the freedom of manually using the API at your will.

2.3.2 Presentations

- django-parler - DjangoCon EU 2014 lightning talk https://speakerdeck.com/vdboor/django-parler-djangocon-eu-2014-lightning-talk

2.3.3 Database schema

django-parler uses a separate table for storing translated fields. Each row stores the content for one language, using a `language_code` column.



The same database layout is used by django-hvad, making a transition to django-parler rather easy.

Advantages:

- Works with existing tools, such as the Django migration framework.
- Unlimited languages can be supported
- Languages can be added on the fly, no database migrations needed.

Disadvantages:

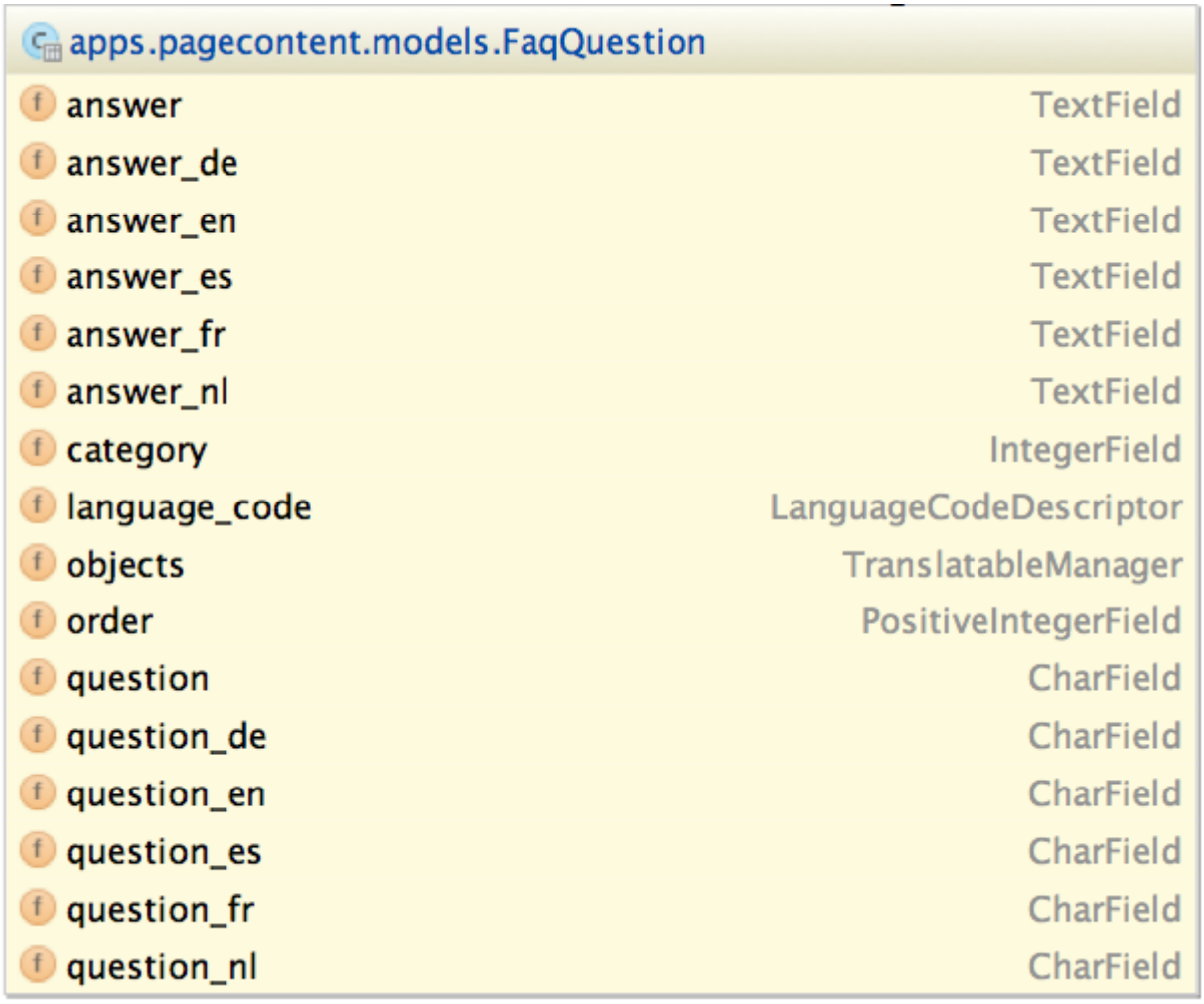
- An extra database query is needed to fetch translated fields.
- Filtering on translated fields should happen in a single `.filter(..)` call.

Solutions:

- The extra database queries are mostly avoided by the caching mechanism, which can store the translated fields in memcached.
- To query all languages, use `.prefetch('translations')` in the ORM query. The prefetched data will be read by django-parler.

Opposite design: django-modeltranslation

The classic solution for writing translatable fields is employed by django-modeltranslation. Each field has a separate column per language.



The advantages are:

- fast reading of all the data, everything is in a single table.
- editing all fields at once is easy.

The disadvantages are:

- The database schema is changed based on the project settings.
- Third party packages can't provide reasonable data migrations for translated fields.
- For projects with a large number of languages, a lot of additional fields will be read with each query,

2.3.4 Package naming

The package name is rather easy to explain; "parler" is French for "to talk".

And for our slogan, watch Dexter's Laboratory episode "The Big Cheese". ;-)

CHAPTER 3

API documentation

3.1 API documentation

3.1.1 parler package

parler.**is_multilingual_project**(*site_id=None*)
Whether the current Django project is configured for multilingual support.

3.1.2 parler.admin module

The BaseTranslatableAdmin class

The TranslatableAdmin class

The TranslatableInlineModelAdmin class

The TranslatableStackedInline class

The TranslatableTabularInline class

The SortedRelatedFieldListFilter class

3.1.3 parler.cache module

django-parler uses caching to avoid fetching model data when it doesn't have to.

These functions are used internally by django-parler to fetch model data. Since all calls to the translation table are routed through our model descriptor fields, cache access and expiry is rather simple to implement.

class parler.cache.**IsMissing**
Bases: object

parler.cache.**get_cached_translated_field**(*instance*, *field_name*, *language_code=None*, *use_fallback=False*)
Fetch an cached field.

parler.cache.**get_cached_translation**(*instance*, *language_code=None*, *related_name=None*, *use_fallback=False*)
Fetch an cached translation.

parler.cache.**get_object_cache_keys**(*instance*)
Return the cache keys associated with an object.

parler.cache.**get_translation_cache_key**(*translated_model*, *master_id*, *language_code*)
The low-level function to get the cache key for a translation.

parler.cache.**is_missing**(*value*)
Check whether the returned value indicates there is no data for the language.

3.1.4 parler.fields module

All fields that are attached to the models.

The core design of django-parler is to attach descriptor fields to the shared model, which then proxies the get/set calls to the translated model.

The *TranslatedField* objects are automatically added to the shared model, but may be added explicitly as well. This also allows to set the `any_language` configuration option. It's also useful for abstract models; add a *TranslatedField* to indicate that the derived model is expected to provide that translatable field.

The `TranslatedField` class

class parler.fields.**TranslatedField**(*any_language=False*)
Proxy field attached to a model.

The field is automatically added to the shared model. However, this can be assigned manually to be more explicit, or to pass the `any_language` value. The `any_language=True` option causes the attribute to always return a translated value, even when the current language and fallback are missing. This can be useful for "title" attributes for example.

Example:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFieldsModel

class MyModel(TranslatableModel):
    title = TranslatedField(any_language=True)  # Add with any-fallback support
    slug = TranslatedField()                    # Optional, but explicitly␣
↪mentioned


class MyModelTranslation(TranslatedFieldsModel):
    # Manual model class:
    master = models.ForeignKey(MyModel, related_name='translations', null=True)
    title = models.CharField("Title", max_length=200)
    slug = models.SlugField("Slug")
```

3.1.5 parler.forms module

The `TranslatableModelForm` class

class parler.forms.**TranslatableModelForm**(**args*, ***kwargs*)
The model form to use for translated models.

The `TranslatableModelFormMixin` class

parler.forms.**TranslatableModelFormMixin**
alias of parler.forms.BaseTranslatableModelForm

The `TranslatedField` class

class parler.forms.**TranslatedField**(***kwargs*)
A wrapper for a translated form field.

This wrapper can be used to declare translated fields on the form, e.g.

```
class MyForm(TranslatableModelForm):
    title = TranslatedField()
    slug = TranslatedField()

    description = TranslatedField(form_class=forms.CharField, widget=TinyMCE)
```

The `TranslatableBaseInlineFormSet` class

class parler.forms.**TranslatableBaseInlineFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)
The formset base for creating inlines with translatable models.

3.1.6 parler.managers module

Custom generic managers

The `TranslatableManager` class

class parler.managers.**TranslatableManager**
The manager class which ensures the enhanced TranslatableQuerySet object is used.

active_translations(*language_code=None*, ***translated_fields*)
Only return objects which are translated, or have a fallback that should be displayed.

Typically that's the currently active language and fallback language. This should be combined with `.distinct()`.

When `hide_untranslated = True`, only the currently active language will be returned.

get_query_set()
Returns a new QuerySet object. Subclasses can override this method to easily customize the behavior of the Manager.

get_queryset()
Returns a new QuerySet object. Subclasses can override this method to easily customize the behavior of the Manager.

language(*language_code=None*)
Set the language code to assign to objects retrieved using this Manager.

queryset_class
alias of *TranslatableQuerySet*

translated(**language_codes*, ***translated_fields*)
Only return objects which are translated in the given languages.

NOTE: due to Django ORM limitations, this method can't be combined with other filters that access the translated fields. As such, query the fields in one filter:

```
qs.translated('en', name="Cheese Omelette")
```

This will query the translated model for the `name` field.

The `TranslatableQuerySet` class

class parler.managers.**TranslatableQuerySet**(**args*, ***kwargs*)
An enhancement of the QuerySet which sets the objects language before they are returned.

When using this class in combination with *django-polymorphic*, make sure this class is first in the chain of inherited classes.

active_translations(*language_code=None*, ***translated_fields*)
Only return objects which are translated, or have a fallback that should be displayed.

Typically that's the currently active language and fallback language. This should be combined with `.distinct()`.

When `hide_untranslated = True`, only the currently active language will be returned.

create(***kwargs*)
Creates a new object with the given kwargs, saving it to the database and returning the created object.

language(*language_code=None*)
Set the language code to assign to objects retrieved using this QuerySet.

translated(**language_codes*, ***translated_fields*)
Only return translated objects which of the given languages.

When no language codes are given, only the currently active language is returned.

Note: Due to Django ORM limitations, this method can't be combined with other filters that access the translated fields. As such, query the fields in one filter:

```
qs.translated('en', name="Cheese Omelette")
```

This will query the translated model for the `name` field.

3.1.7 parler.models module

The models and fields for translation support.

The default is to use the *TranslatedFields* class in the model, like:

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible
from parler.models import TranslatableModel, TranslatedFields


@python_2_unicode_compatible
class MyModel(TranslatableModel):
    translations = TranslatedFields(
        title = models.CharField(_("Title"), max_length=200)
    )

    class Meta:
        verbose_name = _("MyModel")

    def __str__(self):
        return self.title
```

It's also possible to create the translated fields model manually:

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible
from parler.models import TranslatableModel, TranslatedFieldsModel
from parler.fields import TranslatedField


class MyModel(TranslatableModel):
    title = TranslatedField()  # Optional, explicitly mention the field

    class Meta:
        verbose_name = _("MyModel")

    def __str__(self):
        return self.title


class MyModelTranslation(TranslatedFieldsModel):
    master = models.ForeignKey(MyModel, related_name='translations', null=True)
    title = models.CharField(_("Title"), max_length=200)

    class Meta:
        verbose_name = _("MyModel translation")
```

This has the same effect, but also allows to to override the `save()` method, or add new methods yourself.

The translated model is compatible with django-hvad, making the transition between both projects relatively easy. The manager and queryset objects of django-parler can work together with django-mptt and django-polymorphic.

The `TranslatableModel` model

class parler.models.**TranslatableModel**(**args*, ***kwargs*)

Base model class to handle translations.

All translatable fields will appear on this model, proxying the calls to the *TranslatedFieldsModel*.

The `TranslatedFields` class

class parler.models.**TranslatedFields**(*meta=None*, ***fields*)

Wrapper class to define translated fields on a model.

The field name becomes the related name of the *`TranslatedFieldsModel`* subclass.

Example:

```
from django.db import models
from parler.models import TranslatableModel, TranslatedFields

class MyModel(TranslatableModel):
    translations = TranslatedFields(
        title = models.CharField("Title", max_length=200)
    )
```

When the class is initialized, the attribute will point to a `ForeignRelatedObjectsDescriptor` object. Hence, accessing `MyModel.translations.related.related_model` returns the original model via the `django.db.models.related.RelatedObject` class.

Parameters **meta** – A dictionary of *Meta* options, passed to the *`TranslatedFieldsModel`* instance.

Example:

```
class MyModel(TranslatableModel):
    translations = TranslatedFields(
        title = models.CharField("Title", max_length=200),
        slug = models.SlugField("Slug"),
        meta = {'unique_together': [('language_code', 'slug')]},
    )
```

The `TranslatedFieldsModel` model

class parler.models.**TranslatedFieldsModel**(**args*, ***kwargs*)

The `TranslatedFieldsModelBase` metaclass

class parler.models.**TranslatedFieldsModelBase**

Meta-class for the translated fields model.

It performs the following steps:

- It validates the 'master' field, in case it's added manually.
- It tells the original model to use this model for translations.
- It adds the proxy attributes to the shared model.

The `TranslationDoesNotExist` exception

class parler.models.**TranslationDoesNotExist**

A tagging interface to detect missing translations. The exception inherits from `AttributeError` to reflect what is actually happening. Therefore it also causes the templates to handle the missing attributes silently, which is very useful in the admin for example. The exception also inherits from `ObjectDoesNotExist`, so any code that checks for this can deal with missing translations out of the box.

This class is also used in the `DoesNotExist` object on the translated model, which inherits from:

- this class
- the `sharedmodel.DoesNotExist` class
- the original `translatedmodel.DoesNotExist` class.

This makes sure that the regular code flow is decently handled by existing exception handlers.

3.1.8 parler.signals module

The signals exist to make it easier to connect to automatically generated translation models.

To run additional code after saving, consider overwriting `save_translation()` instead. Use the signals as last resort, or to maintain separation of concerns.

pre_translation_init

parler.signals.**pre_translation_init**

This is called when the translated model is initialized, like `pre_init`.

Arguments sent with this signal:

sender As above: the model class that just had an instance created.

instance The actual translated model that's just been created.

args Any arguments passed to the model.

kwargs Any keyword arguments passed to the model.

post_translation_init

parler.signals.**post_translation_init**

This is called when the translated model has been initialized, like `post_init`.

Arguments sent with this signal:

sender As above: the model class that just had an instance created.

instance The actual translated model that's just been created.

pre_translation_save

parler.signals.**pre_translation_save**

This is called before the translated model is saved, like `pre_save`.

Arguments sent with this signal:

sender The model class.

instance The actual translated model instance being saved.

raw `True` when the model is being created by a fixture.

using The database alias being used

post_translation_save

parler.signals.**post_translation_save**

This is called after the translated model has been saved, like `post_save`.

Arguments sent with this signal:

sender The model class.

instance The actual translated model instance being saved.

raw `True` when the model is being created by a fixture.

using The database alias being used

pre_translation_delete

parler.signals.**pre_translation_delete**

This is called before the translated model is deleted, like `pre_delete`.

Arguments sent with this signal:

sender The model class.

instance The actual translated model instance being deleted.

using The database alias being used

post_translation_delete

parler.signals.**post_translation_delete**

This is called after the translated model has been deleted, like `post_delete`.

Arguments sent with this signal:

sender The model class.

instance The actual translated model instance being deleted.

using The database alias being used

3.1.9 parler.utils package

Utility functions to handle language codes and settings.

parler.utils.**normalize_language_code**(*code*)
Undo the differences between language code notations

parler.utils.**is_supported_django_language**(*language_code*)
Return whether a language code is supported.

parler.utils.**get_language_title**(*language_code*)
Return the verbose_name for a language code.

Fallback to language_code if language is not found in settings.

parler.utils.**get_language_settings**(*language_code*, *site_id=None*)
Return the language settings for the current site

parler.utils.**get_active_language_choices**(*language_code=None*)
Find out which translations should be visible in the site. It returns a tuple with either a single choice (the current language), or a tuple with the current language + fallback language.

parler.utils.**is_multilingual_project**(*site_id=None*)
Whether the current Django project is configured for multilingual support.

Submodules:

parler.utils.conf module

The configuration wrappers that are used for *PARLER_LANGUAGES*.

class parler.utils.conf.**LanguagesSetting**
Bases: dict

This is the actual object type of the *PARLER_LANGUAGES* setting. Besides the regular dict behavior, it also adds some additional methods.

get_active_choices(*language_code=None*, *site_id=None*)
Find out which translations should be visible in the site. It returns a list with either a single choice (the current language), or a list with the current language + fallback language.

get_default_language()
Return the default language.

get_fallback_language(*language_code=None*, *site_id=None*)
Find out what the fallback language is for a given language choice.

Deprecated since version 1.5: Use *get_fallback_languages()* instead.

get_fallback_languages(*language_code=None*, *site_id=None*)
Find out what the fallback language is for a given language choice.

get_first_language(*site_id=None*)
Return the first language for the current site. This can be used for user interfaces, where the languages are displayed in tabs.

get_language(*language_code*, *site_id=None*)
Return the language settings for the current site

This function can be used with other settings variables to support modules which create their own variation of the PARLER_LANGUAGES setting. For an example, see add_default_language_settings().

parler.utils.conf.**add_default_language_settings**(*languages_list*, *var_name='PARLER_LANGUAGES'*, ***extra_defaults*)
Apply extra defaults to the language settings. This function can also be used by other packages to create their own variation of PARLER_LANGUAGES with extra fields. For example:

```
from django.conf import settings
from parler import appsettings as parler_appsettings

# Create local names, which are based on the global parler settings
MYAPP_DEFAULT_LANGUAGE_CODE = getattr(settings, 'MYAPP_DEFAULT_LANGUAGE_CODE',␣
↪parler_appsettings.PARLER_DEFAULT_LANGUAGE_CODE)
MYAPP_LANGUAGES = getattr(settings, 'MYAPP_LANGUAGES', parler_appsettings.PARLER_
↪LANGUAGES)

# Apply the defaults to the languages
```

(continues on next page)

(continued from previous page)

```
MYAPP_LANGUAGES = parler_appsettings.add_default_language_settings(MYAPP_
↪LANGUAGES, 'MYAPP_LANGUAGES',
    code=MYAPP_DEFAULT_LANGUAGE_CODE,
    fallback=MYAPP_DEFAULT_LANGUAGE_CODE,
    hide_untranslated=False
)
```

The returned object will be an *LanguagesSetting* object, which adds additional methods to the dict object.

Parameters

- **languages_list** – The settings, in *PARLER_LANGUAGES* format.
- **var_name** – The name of your variable, for debugging output.
- **extra_defaults** – Any defaults to override in the languages_list['default'] section, e.g. code, fallback, hide_untranslated.

Returns The updated languages_list with all defaults applied to all sections.

Return type *LanguagesSetting*

parler.utils.conf.**get_parler_languages_from_django_cms**(*cms_languages=None*)

Converts django CMS' setting CMS_LANGUAGES into PARLER_LANGUAGES. Since CMS_LANGUAGES is a strict superset of PARLER_LANGUAGES, we do a bit of cleansing to remove irrelevant items.

parler.utils.context module

Context managers for temporary switching the language.

class parler.utils.context.**smart_override**(*language_code*)

This is a smarter version of translation.override which avoids switching the language if there is no change to make. This method can be used in place of translation.override:

```
with smart_override(self.get_current_language()):
    return reverse('myobject-details', args=(self.id,))
```

This makes sure that any URLs wrapped in i18n_patterns() will receive the correct language code prefix. When the URL also contains translated fields (e.g. a slug), use *switch_language* instead.

class parler.utils.context.**switch_language**(*object*, *language_code=None*)

A contextmanager to switch the translation of an object.

It changes both the translation language, and object language temporary.

This context manager can be used to switch the Django translations to the current object language. It can also be used to render objects in a different language:

```
with switch_language(object, 'nl'):
    print object.title
```

This is particularly useful for the get_absolute_url() function. By using this context manager, the object language will be identical to the current Django language.

```
def get_absolute_url(self):
    with switch_language(self):
        return reverse('myobject-details', args=(self.slug,))
```

Note: When the object is shared between threads, this is not thread-safe. Use `safe_translation_getter()` instead to read the specific field.

3.1.10 parler.views module

The views provide high-level utilities to integrate translation support into other projects.

The following mixins are available:

- *ViewUrlMixin* - provide a `get_view_url` for the *{% get_translated_url %}* template tag.
- *TranslatableSlugMixin* - enrich the `DetailView` to support translatable slugs.
- *LanguageChoiceMixin* - add `?language=xx` support to a view (e.g. for editing).
- *TranslatableModelFormMixin* - add support for translatable forms, e.g. for creating/updating objects.

The following views are available:

- *TranslatableCreateView* - The `CreateView` with *TranslatableModelFormMixin* support.
- *TranslatableUpdateView* - The `UpdateView` with *TranslatableModelFormMixin* support.

The `ViewUrlMixin` class

class parler.views.**ViewUrlMixin**

Provide a `view.get_view_url` method in the template.

This tells the template what the exact canonical URL should be of a view. The *{% get_translated_url %}* template tag uses this to find the proper translated URL of the current page.

Typically, setting the *view_url_name* just works:

```
class ArticleListView(ViewUrlMixin, ListView):
    view_url_name = 'article:list'
```

The *get_view_url()* will use the *view_url_name* together with `view.args` and `view.kwargs` construct the URL. When some arguments are translated (e.g. a slug), the *get_view_url()* can be overwritten to generate the proper URL:

```
from parler.views import ViewUrlMixin, TranslatableUpdateView
from parler.utils.context import switch_language

class ArticleEditView(ViewUrlMixin, TranslatableUpdateView):
    view_url_name = 'article:edit'

    def get_view_url(self):
        with switch_language(self.object, get_language()):
            return reverse(self.view_url_name, kwargs={'slug': self.object.slug})
```

get_view_url()

This method is used by the `get_translated_url` template tag.

By default, it uses the *view_url_name* to generate an URL. When the URL `args` and `kwargs` are translatable, override this function instead to generate the proper URL.

view_url_name = None
The default view name used by *get_view_url()*, which should correspond with the view name in the URLConf.

The TranslatableSlugMixin class

class parler.views.**TranslatableSlugMixin**
An enhancement for the DetailView to deal with translated slugs. This view makes sure that:

- The object is fetched in the proper translation.
- The slug field is read from the translation model, instead of the shared model.
- Fallback languages are handled.
- Objects are not accidentally displayed in their fallback slug, but redirect to the translated slug.

Example:

```
class ArticleDetailView(TranslatableSlugMixin, DetailView):
    model = Article
    template_name = 'article/details.html'
```

get_language()
Define the language of the current view, defaults to the active language.

get_language_choices()
Define the language choices for the view, defaults to the defined settings.

get_object(*queryset=None*)
Fetch the object using a translated slug.

get_translated_filters(*slug*)
Allow passing other filters for translated fields.

The LanguageChoiceMixin class

class parler.views.**LanguageChoiceMixin**
Mixin to add language selection support to class based views, particularly create and update views. It adds support for the ?language=.. parameter in the query string, and tabs in the context.

get_current_language()
Return the current language for the currently displayed object fields. This reads self.object.get_current_language() and falls back to *get_language()*.

get_default_language(*object=None*)
Return the default language to use, if no language parameter is given. By default, it uses the default parler-language.

get_language()
Get the language parameter from the current request.

get_language_tabs()
Determine the language tabs to show.

get_object(*queryset=None*)
Assign the language for the retrieved object.

The `TranslatableModelFormMixin` class

class parler.views.**TranslatableModelFormMixin**
Mixin to add translation support to class based views.

For example, adding translation support to django-oscar:

```
from oscar.apps.dashboard.catalogue import views as oscar_views
from parler.views import TranslatableModelFormMixin

class ProductCreateUpdateView(TranslatableModelFormMixin, oscar_views.
↪ProductCreateUpdateView):
    pass
```

get_form_class()
Return a `TranslatableModelForm` by default if no form_class is set.

get_form_kwargs()
Pass the current language to the form.

The `TranslatableCreateView` class

class parler.views.**TranslatableCreateView**(***kwargs*)
Create view that supports translated models. This is a mix of the *`TranslatableModelFormMixin`* and Django's `CreateView`.

The `TranslatableUpdateView` class

class parler.views.**TranslatableUpdateView**(***kwargs*)
Update view that supports translated models. This is a mix of the *`TranslatableModelFormMixin`* and Django's `UpdateView`.

3.1.11 parler.widgets module

These widgets perform sorting on the choices within Python. This is useful when sorting is hard to due translated fields, for example:

- the ORM can't sort it.
- the ordering depends on `ugettext()` output.
- the model `__unicode__()` value depends on translated fields.

Use them like any regular form widget:

```
from django import forms
from parler.widgets import SortedSelect

class MyModelForm(forms.ModelForm):
    class Meta:
        # Make sure translated choices are sorted.
        model = MyModel
        widgets = {
            'preferred_language': SortedSelect,
            'country': SortedSelect,
        }
```

The `SortedSelect` class

class parler.widgets.**SortedSelect**(*attrs=None*, *choices=()*)
A select box which sorts it's options.

The `SortedSelectMultiple` class

class parler.widgets.**SortedSelectMultiple**(*attrs=None*, *choices=()*)
A multiple-select box which sorts it's options.

The `SortedCheckboxSelectMultiple` class

class parler.widgets.**SortedCheckboxSelectMultiple**(*attrs=None*, *choices=()*)
A checkbox group with sorted choices.

3.2 Changelog

3.2.1 Changes in 1.9.2 (2018-02-12)

- Fixed Django 2.0 `contribute_to_class()` call signature.
- Fixed "Save and add another" button redirect when using different admin sites.
- Fixed import location of `mark_safe()`.

3.2.2 Changes in 1.9.1 (2017-12-06)

- Fixed HTML output in Django 2.0 for the the `language_column` and `all_languages_column` fields in the Django admin.

3.2.3 Changes in 1.9 (2017-12-04)

- Added Django 2.0 support.
- Fixed `get_or_create()` call when no defaults are given.

3.2.4 Changes in 1.8.1 (2017-11-20)

- Fixed checkes for missing fallback languages (`IsMissing` sentinel value leaked through caching)
- Fixed preserving the language tab in the admin.
- Fixed `get_or_create()` call.

3.2.5 Changes in 1.8 (2017-06-20)

- Dropped Django 1.5, 1.6 and Python 2.6 support.
- Fixed Django 1.10 / 1.11 support:
  - Fix `.language('xx').get()` usage.
  - Fix models construction via `Model(**kwargs)`.
  - Fix test warnings due to tests corrupting the app registry.
- Fix support for `ModelFormMixin.fields` in `TranslatableUpdateView`. Django allows that attribute as alternative to setting a `form_class` manually.

3.2.6 Changes in 1.7 (2016-11-29)

- Added `delete_translation()` API.
- Added `PARLER_DEFAULT_ACTIVATE` setting, which allows to display translated texts in the default language even through `translation.activate()` is not called yet.
- Improve language code validation in forms, allows to enter a language variant.
- Fixed not creating translations when default values were filled in.
- Fixed breadcrumb errors in delete translation view when using django-polymorphic-tree.

3.2.7 Changes in 1.6.5 (2016-07-11)

- Fix `get_translated_url()` when Django uses bytestrings for `QUERY_STRING`.
- Raise `ValidError` when a `TranslatableForm` is initialized with a language code that is not available in `LANGUAGES`.

Backwards compatibility note: An `ValueError` is now raised when forms are initialized with an invalid languae code. If your project relied on invalid language settings, make sure that `LANGAUGE_CODE` and `LANGUAGES` are properly configured.

Rationale: Since the security fix in v1.6.3 (to call the `clean()` method of translated fields), invalid language codes are no longer accepted. The choice was either to passively warn and exclude the language from validation checks, or to raise an error beforehand that the form is used to initialize bad data. It's considered more important to avoid polluted database contents then preserving compatibility, hence this check remains as strict.

3.2.8 Changes in 1.6.4 (2016-06-14)

- Fix calling `clean()` on fields that are not part of the form.
- Fix tab appearance for Django 1.9 and flat theme.
- Fix issues with `__proxy__` field for template names
- Fix attempting to save invalid `None` language when Django translations are not yet initialized.

Note: django-parler models now mandate that a language code is selected; either by calling `model.set_current_language()`, `Model.objects.language()` or activating a gettext environment. The latter always happens in a standard web request, but needs to happen explicitly in management commands. This avoids hard to debug situations where unwanted model changes happen on implicitly selected languages.

3.2.9 Changes in 1.6.3 (2016-05-05)

- **Security notice:** Fixed calling `clean()` on the translations model.
- Fixed error with M2M relations to the translated model.
- Fixed `UnicodeError` in `parler_tags`
- Show warning when translations are not initialized (when using management commands).

3.2.10 Changes in 1.6.2 (2016-03-08)

- Added `TranslatableModelMixin` to handle complex model inheritance issues.
- Fixed tuple/list issues with `fallbacks` option.
- Fixed Python 3 *__str__()* output for `TranslatedFieldsModel`.
- Fixed output for `get_language_title()` when language is not configured.
- Fixed preserving GET args in admin change form view.

3.2.11 Changes in version 1.6.1 (2016-02-11)

- Fix queryset `.dates()` iteration in newer Django versions.
- Fixed Django 1.10 deprecation warnings in the admin.
- Avoided absolute URLs in language tabs.

3.2.12 Changes in version 1.6 (2015-12-29)

- Added Django 1.9 support
- Added support to generate `PARLER_LANGUAGES` from Django CMS' `CMS_LANGUAGES`.
- Improve language variant support, e.g. `fr-ca` can fallback to `fr`, and `de-ch` can fallback to `de`.
- Dropped support for Django 1.4

(also released as 1.6b1 on 2015-12-16)

3.2.13 Changes in version 1.5.1 (2015-10-01)

- Fix handling for non-nullable `ForeignKey` in forms and admin.
- Fix performance of the admin list when `all_languages_column` or `language_column` is added to `list_display` (N-query issue).
- Fix support for other packages that replace the BoundField class in `Form.__get_item__` (namely django-slug-preview).
- Fix editing languages that exist in the database but are not enabled in project settings.
- Fix DeprecationWarning for Django 1.8 in the admin.

3.2.14 Changes in version 1.5 (2015-06-30)

- Added support for multiple fallback languages!
- Added `translatable-field` CSS class to the `<label>` tags of translatable fields.
- Added `{{ field.is_translatable }}` variable.
- Added warning when saving a model without language code set. As of Django 1.8, `get_language()` returns `None` if no language is activated.
- Allow `safe_translation_getter(default=..)` to be a callable.
- Added `all_languages_column`, inspired by aldryn-translation-tools.
- Changed styling of `language_column`, the items are now links to the language tabs.
- Fix caching support, the default timeout was wrongly imported.
- Fix Django 1.4 support for using `request.resolver_match`.
- Fix admin delete translation view when using `prefetch_related('translations')` by default in the managers `get_queryset()` method.
- Fix using prefetched translations in `has_translation()` too.
- Return to change view after deleting a translation.

3.2.15 Changes in version 1.4 (2015-04-13)

- Added Django 1.8 support
- Fix caching when using redis-cache
- Fix handling `update_fields` in `save()` (needed for combining parler with django-mptt 0.7)
- Fix unwanted migration changes in Django 1.6/South for the internal `HideChoicesCharField`.
- Fix overriding get_current_language() / get_form_language() in the `TranslatableModelFormMixin`/`TranslatableCre`

3.2.16 Changes in version 1.3 (2015-03-13)

- Added support for `MyModel.objects.language(..).create(..)`.
- Detect when translatable fields are assigned too early.
- Fix adding `choices=LANGUAGES` to all Django 1.7 migrations.
- Fix missing 404 check in delete-translation view.
- Fix caching for models that have a string value as primary key.
- Fix support for a primary-key value of `0`.
- Fix `get_form_class()` override check for `TranslatableModelFormMixin` for Python 3.
- Fix calling manager methods on related objects in Django 1.4/1.5.
- Improve `{% get_translated_url %}`, using `request.resolver_match` value.
- Fix preserving query-string in `{% get_translated_url %}`, unless an object is explicitly passed.
- Fix supporting removed model fields in `get_cached_translation()`.

3.2.17 Changes in version 1.2.1 (2014-10-31)

- Fixed fetching correct translations when using `prefetch_related()`.

3.2.18 Changes in version 1.2 (2014-10-30)

- Added support for translations on multiple model inheritance levels.
- Added `TranslatableAdmin.get_translation_objects()` API.
- Added `TranslatableModel.create_translation()` API.
- Added `TranslatableModel.get_translation()` API.
- Added `TranslatableModel.get_available_languages(include_unsaved=True)` API.
- **NOTE:** the `TranslationDoesNotExist` exception inherits from `ObjectDoesNotExist` now. Check your exception handlers when upgrading.

3.2.19 Changes in version 1.1.1 (2014-10-14)

- Fix accessing fields using `safe_translation_getter(any_language=True)`
- Fix "dictionary changed size during iteration" in `save_translations()` in Python 3.
- Added `default_permissions=()` for translated models in Django 1.7.

3.2.20 Changes in version 1.1 (2014-09-29)

- Added Django 1.7 compatibility.
- Added `SortedRelatedFieldListFilter` for displaying translated models in the `list_filter`.
- Added `parler.widgets` with `SortedSelect` and friends.
- Fix caching translations in Django 1.6.
- Fix checking `unique_together` on the translated model.
- Fix access to `TranslatableModelForm._current_language` in early `__init__()` code.
- Fix `PARLER_LANGUAGES['default']['fallback']` being overwritten by `PARLER_DEFAULT_LANGUAGE_CODE`.
- Optimized prefetch usage, improves loading of translated models.
- **BACKWARDS INCOMPATIBLE:** The arguments of `get_cached_translated_field()` have changed ordering, `field_name` comes before `language_code` now.

3.2.21 Changes in version 1.0 (2014-07-07)

Released in 1.0b3:

- Added `TranslatableSlugMixin`, to be used for detail views.
- Fixed translated field names in admin `list_display`, added `short_description` to `TranslatedFieldDescriptor`
- Fix internal server errors in `{% get_translated_url %}` for function-based views with class kwargs

- Improved admin layout for `save_on_top=True`.

Released in 1.0b2:

- Fixed missing app_label in cache key, fixes support for multiple models with the same name.
- Fixed "dictionary changed size during iteration" in `save_translations()`

Released in 1.0b1:

- Added `get_translated_url` template tag, to implement language switching easily. This also allows to implement hreflang support for search engines.
- Added a `ViewUrlMixin` so views can tell the template what their exact canonical URL should be.
- Added `TranslatableCreateView` and `TranslatableUpdateView` views, and associated mixins.
- Fix missing "language" GET parmeter for Django 1.6 when filtering in the admin (due to the `_changelist_filters` parameter).
- Support missing *SITE_ID* setting for Django 1.6.

Released in 1.0a1:

- **BACKWARDS INCOMPATIBLE:** updated the model name of the dynamically generated translation models for django-hvad compatibility. This only affects your South migrations. Use `manage.py schemamigration appname --empty "upgrade_to_django_parler10"` to upgrade applications which use `translations = TranslatedFields(..)` in their models.
- Added Python 3 compatibility!
- Added support for `.prefetch('translations')`.
- Added automatic caching of translated objects, use `PARLER_ENABLE_CACHING = False` to disable.
- Added inline tabs support (if the parent object is not translatable).
- Allow `.translated()` and `.active_translations()` to filter on translated fields too.
- Added `language_code` parameter to `safe_translation_getter()`, to fetch a single field in a different language.
- Added `switch_language()` context manager.
- Added `get_fallback_language()` to result of `add_default_language_settings()` function.
- Added partial support for tabs on inlines when the parent object isn't a translated model.
- Make settings.SITE_ID setting optional
- Fix inefficient or unneeded queries, i.e. for new objects.
- Fix supporting different database (using=) arguments.
- Fix list language, always show translated values.
- Fix `is_supported_django_language()` to support dashes too
- Fix ignored `Meta.fields` declaration on forms to exclude all other fields.

3.2.22 Changes in version 0.9.4 (beta)

- Added support for inlines!
- Fix error in Django 1.4 with "Save and continue" button on add view.
- Fix error in `save_translations()` when objects fetched fallback languages.
- Add `save_translation(translation)` method, to easily hook into the `translation.save()` call.
- Added support for empty `translations = TranslatedFields()` declaration.

3.2.23 Changes in version 0.9.3 (beta)

- Support using `TranslatedFieldsModel` with abstract models.
- Added `parler.appsettings.add_default_language_settings()` function.
- Added `TranslatableManager.queryset_class` attribute to easily customize the queryset class.
- Added `TranslatableManager.translated()` method to filter models with a specific translation.
- Added `TranslatableManager.active_translations()` method to filter models which should be displayed.
- Added `TranslatableAdmin.get_form_language()` to access the currently active language.
- Added `hide_untranslated` option to the `PARLER_LANGUAGES` setting.
- Added support for `ModelAdmin.formfield_overrides`.

3.2.24 Changes in version 0.9.2 (beta)

- Added `TranslatedField(any_language=True)` option, which uses any language as fallback in case the currently active language is not available. This is ideally suited for object titles.
- Improved `TranslationDoesNotExist` exception, now inherits from `AttributeError`. This missing translations fail silently in templates (e.g. admin list template)..
- Added unittests
- Fixed Django 1.4 compatibility
- Fixed saving all translations, not only the active one.
- Fix sending `pre_translation_save` signal.
- Fix passing `_current_language` to the model __init__ function.

3.2.25 Changes in version 0.9.1 (beta)

- Added signals to detect translation model init/save/delete operations.
- Added default `TranslatedFieldsModel` `verbose_name`, to improve the delete view.
- Allow using the `TranslatableAdmin` for non-`TranslatableModel` objects (operate as NO-OP).

3.2.26 Changes in version 0.9 (beta)

- First version, based on intermediate work in django-fluent-pages. Integrating django-hvad turned out to be very complex, hence this app was developped instead.

CHAPTER 4

Roadmap

The following features are on the radar for future releases:

- Multi-level model inheritance support
- Improve query usage, e.g. by adding “Prefetch” objects.

Please contribute your improvements or work on these area’s!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index