django-parler Documentation

Release 1.2.1

Diederik van der Boor and contributors

December 30, 2014

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 default language.
- 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 possibly the fallback language, 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 language 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 language is 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': {
        'fallback': '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_LANUAGES['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': {
        'fallback': '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.

`fallback` The fallback language for the entry

`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': {
        'fallback': '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
```

If needed, caching can be disabled. This is likely not needed.

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 %}
        {% get_translated_url lang_code as tr_url %}
        {% if tr_url %}<li{% if lang_code == LANGUAGE_CODE %} class="is-selected"{% endif %}><a href=
    {% 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).

CHAPTER 2

In depth topics

2.1 Advanced usage patterns

2.1.1 Translations without fallback languages

When a translation is missing, the fallback language is used. However, when an object has no fallback language, 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 slug, 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 South, use: `manage.py schemamigration myapp --auto "add_translation_model"`

Step 2: Copy the data

Within the data migration, copy the existing data:

Using Django

Create an empty migration:

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

And use it to move the data:

```
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() in South.

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
            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.

Using South

With South, create a data migration:

```
manage.py datamigration myapp "migrate_translatable_fields"
```

The logic is identical, only the way for receiving the ORM models differs:

```
class Migration(DataMigration):

    def forwards(self, orm):
        MyModel = orm['myapp.MyModel']
        MyModelTranslation = orm['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(self, orm):
        # Convert all fields back to the single-language table.
        MyModel = orm['myapp.MyModel']
        MyModelTranslation = orm['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() in South.


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
            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()
```

The forwards method can also be implemented in raw SQL:

```
class Migration(DataMigration):

    def forwards(self, orm):
        db.execute(
            'INSERT INTO myapp_mymodel_translation(name, language_code, master_id)'
            ' SELECT name, _cached_url, %s, id FROM myapp_mymodel',
            [settings.LANGUAGE_CODE]
        )
```

Note: Be careful which language is used to migrate the existing data. In this example, the `backwards()` 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 South, use: `manage.py schemamigration myapp --auto "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-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.

Note: You need at least django-polymorphic >= 0.5.6 in order to get this working.

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

    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(TranslatableModel, Product):
    # 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)
```

2.1.6 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 %}
```

2.1.7 Integration with django-rest-framework

To integrate the translated fields in django-rest-framework, the `parler.contrib.rest_framework` 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=

    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.contrib.rest_framework import TranslatableModelSerializer, TranslatedFieldsField
from myapp.models import Country

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

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

2.1.8 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.

2.1.9 Disabling caching

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

2.1.10 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.

2.1.11 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

MYCMS_DEFAULT_LANGUAGE_CODE = getattr(settings, 'MYCMS_DEFAULT_LANGUAGE_CODE', FLUENT_DEFAULT_LANGUAG
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,
    fallback=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_language()
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 1.4, 1.5, 1.6 and 1.7
- Python versions 2.6, 2.7, 3.3 and 3.4

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

2.2.5 Using `fieldsets` in Django 1.4

When using Django 1.4, there is a small tweak you'll have to make in the admin. Instead of using `fieldsets`, use `declared_fieldsets` on the `ModelAdmin` definition.

The Django 1.4 admin validation doesn't actually check the form fields, but only checks whether the fields exist in the model - which they obviously don't. Using `declared_fieldsets` instead of `fieldsets` circumvents this check.

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.

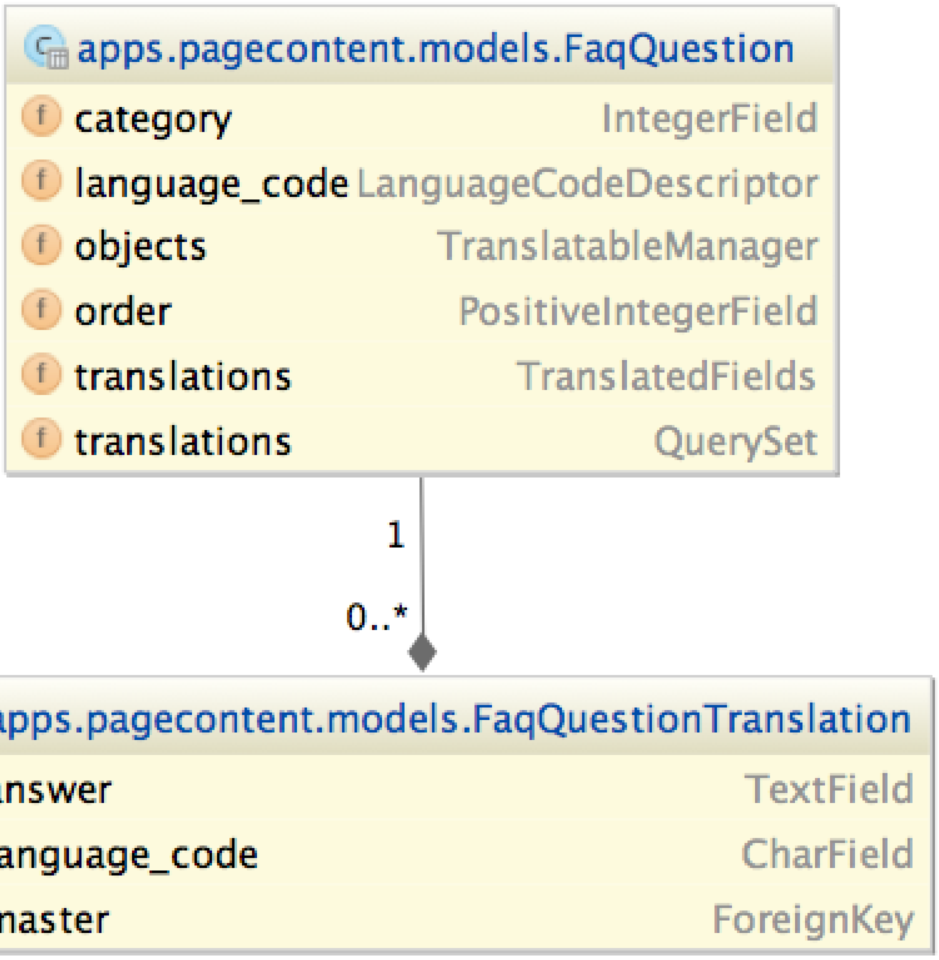
Added on top of that, the API-suger is provided, similar to what django-hvad has. It's possible to create the translations model manually, or let it be created dynamically when using the `TranslatedFields` field. This is to make your life easier - without loosing the freedom of manually using the API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


apps.pagecontent.models.FaqQuestion
category IntegerField
language_code LanguageCodeDescriptor
objects TranslatableManager
order PositiveIntegerField
translations TranslatedFields
translations QuerySet
1
0..*
apps.pagecontent.models.FaqQuestionTranslation
answer TextField
language_code CharField
master ForeignKey

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

Advantages:

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

Disadvantages:

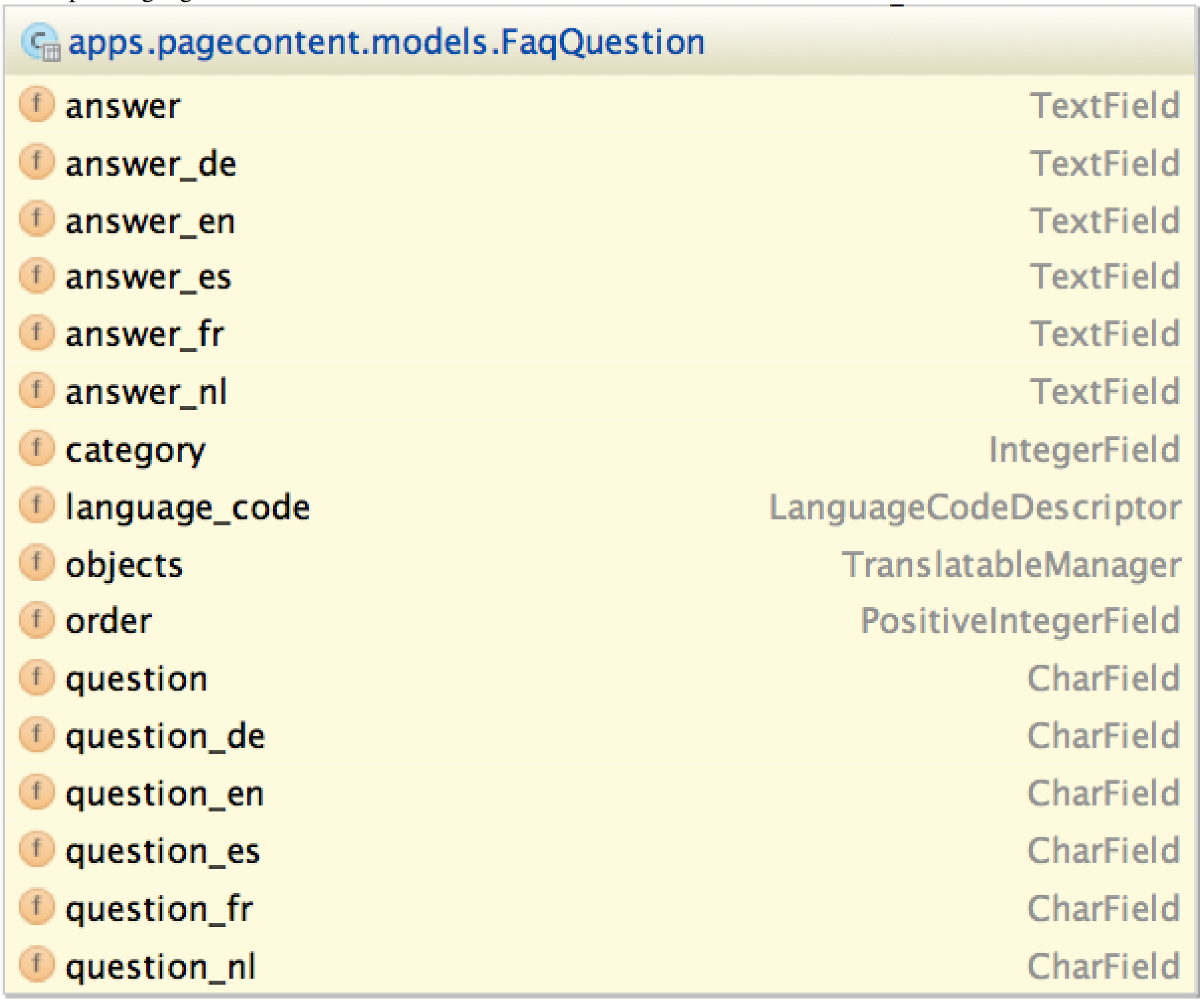
- An extra database query is needed to fetch translated fields.
- Filtering on translated fields should happen in a single `.filter(..)` call.

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.



apps.pagecontent.models.FaqQuestion

answer	TextField
answer_de	TextField
answer_en	TextField
answer_es	TextField
answer_fr	TextField
answer_nl	TextField
category	IntegerField
language_code	LanguageCodeDescriptor
objects	TranslatableManager
order	PositiveIntegerField
question	CharField
question_de	CharField
question_en	CharField
question_es	CharField
question_fr	CharField
question_nl	CharField

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 South 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

Translation support for admin forms.

django-parler provides the following classes:

- Model support: `TranslatableAdmin`.
- Inline support: `TranslatableInlineModelAdmin`, `TranslatableStackedInline`, `TranslatableTabularInline`.
- Utilities: `SortedRelatedFieldListFilter`.

Admin classes can be created as expected:

```
from django.contrib import admin
from parler.admin import TranslatableAdmin
from myapp.models import Project

class ProjectAdmin(TranslatableAdmin):
    list_display = ('title', 'status')
    fieldsets = (
        (None, {
            'fields': ('title', 'status'),
        }),
    )

admin.site.register(Project, ProjectAdmin)
```

All translated fields can be used in the `list_display` and `fieldsets` like normal fields.

While almost every admin feature just works, there are a few special cases to take care of:

- The `search_fields` needs the actual ORM fields.
- The `prepopulated_fields` needs to be replaced with a call to `get_prepopulated_fields()`.

See the *admin compatibility page* for details.

The `BaseTranslatableAdmin` class

class `parler.admin.`**`BaseTranslatableAdmin`**

The shared code between the regular model admin and inline classes.

`form`

The form to use for the model.

alias of `TranslatableModelForm`

`get_form_language`(*request*, *obj=None*)

Return the current language for the currently displayed object fields.

`get_language_tabs`(*request*, *obj*, *available_languages*, *css_class=None*)

Determine the language tabs to show.

`get_queryset`(*request*)

Make sure the current language is selected.

`get_queryset_language`(*request*)

Return the language to use in the queryset.

`query_language_key` **= u'language'**

The URL parameter for the language value.

`queryset`(*request*)

Make sure the current language is selected.

The `TranslatableAdmin` class

class `parler.admin.`**`TranslatableAdmin`**(*model*, *admin_site*)

Base class for translated admins.

This class also works as regular admin for non TranslatableModel objects. When using this class with a non-TranslatableModel, all operations effectively become a NO-OP.

`change_form_template`

Dynamic property to support transition to regular models.

This automatically picks `admin/parler/change_form.html` when the admin uses a translatable model.

`delete_inline_translations` **= True**

Whether translations of inlines should also be deleted when deleting a translation.

`delete_model_translation`(*request*, *translation*)

Hook for deleting a translation. This calls `get_translation_objects()` to collect all related objects for the translation. By default, that includes the translations for inline objects.

`delete_translation`(**args*, ***kwargs*)

The 'delete translation' admin view for this model.

`deletion_not_allowed`(*request*, *obj*, *language_code*)

Deletion-not-allowed view.

`get_available_languages`(*obj*)

Fetching the available languages as queryset.

get_change_form_base_template()
Determine what the actual *change_form_template* should be.

get_form(*request*, *obj=None*, ***kwargs*)
Pass the current language to the form.

get_language_short_title(*language_code*)
Hook for allowing to change the title in the `language_column()` of the list_display.

get_object(*request*, *object_id*)
Make sure the object is fetched in the correct language.

get_translation_objects(*request*, *language_code*, *obj=None*, *inlines=True*)
Return all objects that should be deleted when a translation is deleted. This method can yield all QuerySet objects or lists for the objects.

get_urls()
Add a delete-translation view.

language_column(*object*)
The language column which can be included in the `list_display`.

render_change_form(*request*, *context*, *add=False*, *change=False*, *form_url=u''*, *obj=None*)
Insert the language tabs.

The `TranslatableInlineModelAdmin` class

class parler.admin.**TranslatableInlineModelAdmin**(*parent_model*, *admin_site*)
Base class for inline models.

form
The form to use.

alias of `TranslatableModelForm`

formset
The formset to use.

alias of `TranslatableBaseInlineFormSet`

get_available_languages(*obj*, *formset*)
Fetching the available inline languages as queryset.

get_form_language(*request*, *obj=None*)
Return the current language for the currently displayed object fields.

get_formset(*request*, *obj=None*, ***kwargs*)
Return the formset, and provide the language information to the formset.

inline_tabs
Whether to show inline tabs, can be set as attribute on the inline.

The `TranslatableStackedInline` class

class parler.admin.**TranslatableStackedInline**(*parent_model*, *admin_site*)
The inline class for stacked layout.

The `TranslatableTabularInline` class

class `parler.admin.`**`TranslatableTabularInline`**(*parent_model*, *admin_site*)

The inline class for tabular layout.

The `SortedRelatedFieldListFilter` class

class `parler.admin.`**`SortedRelatedFieldListFilter`**(**args*, ***kwargs*)

Override the standard `RelatedFieldListFilter`, to sort the values after rendering their `__unicode__()` values. This can be used for translated models, which are difficult to sort beforehand. Usage:

```
from django.contrib import admin
from parler.admin import SortedRelatedFieldListFilter

class MyAdmin(admin.ModelAdmin):

    list_filter = (
        ('related_field_name', SortedRelatedFieldListFilter),
    )
```

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.

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

class `parler.forms.`**`TranslatableModelFormMixin`**(**args*, ***kwargs*)

The base methods added to `TranslatableModelForm` to fetch and store translated fields.

`save_translated_fields`()

Save all translated fields.

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.

`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.

`iterator`()

Overwritten iterator which will set the current language before returning the 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 parler.models import TranslatableModel, TranslatedFields


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

    class Meta:
        verbose_name = _("MyModel")

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

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:
        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`.

`create_translation`(*language_code*, ***fields*)

Add a translation to the model.

The `save_translations()` function is called afterwards.

The object will be saved immediately, similar to calling `create()` or `create()` on related fields.

`get_available_languages`(*related_name=None*, *include_unsaved=False*)

Return the language codes of all translated variations.

`get_current_language`()

Get the current language.

`get_fallback_language`()

Return the fallback language code, which is used in case there is no translation for the currently active language.

`get_translation`(*language_code*, *related_name=None*)

Fetch the translated model

`has_translation`(*language_code=None*, *related_name=None*)

Return whether a translation for the given language exists. Defaults to the current language code.

`safe_translation_getter`(*field*, *default=None*, *language_code=None*, *any_language=False*)

Fetch a translated property, and return a default value when both the translation and fallback language are missing.

When `any_language=True` is used, the function also looks into other languages to find a suitable value. This feature can be useful for "title" attributes for example, to make sure there is at least something being displayed. Also consider using `field = TranslatedField(any_language=True)` in the model itself, to make this behavior the default for the given field.

`save_translation`(*translation*, **args*, ***kwargs*)

Save the translation when it's modified, or unsaved.

Note: When a derived model provides additional translated fields, this method receives both the original and extended translation. To distinguish between both objects, check for `translation.related_name`.

Parameters

- **translation** (*TranslatedFieldsModel*) – The translation
- **args** – Any custom arguments to pass to `save()`.

- **kwargs** – Any custom arguments to pass to `save()`.

`save_translations`(**args*, ***kwargs*)

The method to save all translations. This can be overwritten to implement any custom additions. This method calls `save_translation()` for every fetched language.

Parameters

- **args** – Any custom arguments to pass to `save()`.
- **kwargs** – Any custom arguments to pass to `save()`.

`set_current_language`(*language_code*, *initialize=False*)

Switch the currently activate language of the object.

`validate_unique`(*exclude=None*)

Also validate the unique_together of the translated model.

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.model` returns the original model via the `django.db.models.related.RelatedObject` class.

The `TranslatedFieldsModel` model

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

Base class for the model that holds the translated fields.

classmethod `contribute_translations`(*shared_model*)

Add the proxy attributes to the shared model.

`is_empty`

True when there are no translated fields.

`is_modified`

Tell whether the object content is modified since fetching it.

`master` = None

The mandatory Foreign key field to the shared model.

`related_name`

Returns the related name that this model is known at in the shared model.

shared_model
Returns the shared model this model is linked to.

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.

`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 tuple with either a single choice (the current language), or a tuple 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.

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_appsetting
MYAPP_LANGUAGES = getattr(settings, 'MYAPP_LANGUAGES', parler_appsettings.PARLER_LANGUAGES)

# Apply the defaults to the languages
MYAPP_LANGUAGES = parler_appsettings.add_default_language_settings(MYAPP_LANGUAGES, 'MYAPP_LANGU
    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.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.

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_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.

get_form_language()
Return the current language for the currently displayed object fields.

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.

3.1.12 parler.contrib.rest_framework module

This package provides support for integrating translatable fields into *django-rest-framework*.

The `TranslatedFieldsField` model

class `parler.contrib.rest_framework.`**`TranslatedFieldsField`**(**args*, ***kwargs*)

Exposing translated fields for a TranslatableModel in REST style.

`from_native`(*data*, *files=None*)

Deserialize primitives -> objects.

`to_native`(*value*)

Serialize to REST format.

The `TranslatableModelSerializer` class

class `parler.contrib.rest_framework.`**`TranslatableModelSerializer`**(*instance=None*, *data=None*, *files=None*, *context=None*, *partial=False*, *many=False*, *allow_add_remove=False*, ***kwargs*)

Serializer that makes sure that translations from the `TranslatedFieldsField` are properly saved.

It should be used instead of the regular `ModelSerializer`.

`save_object`(*obj*, ***kwargs*)

Extract the translations, store these into the django-parler model data.

3.2 Changelog

3.2.1 Changes in development (git version)

- Added `parler.contrib.rest_framework` package for django-rest-framework integration.
- Added support for `MyModel.objects.language(..).create(..)`.
- Detect when translatable fields are assigned too early.
- 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.

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

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

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

- Added support for translations on mutlple 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.2.8 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`.

3.2.9 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.

3.2.10 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).

3.2.11 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*