django-modeltrans

Release 0.7.5

Aug 03, 2023

Documentation

1 Table of contents 3

Index 21

- Uses one `django.contrib.postgres.JSONField` (PostgreSQL JSONB field) for every record.
- Django 1.11 and 2.0 (with their supported python versions)
- PostgreSQL >= 9.5 and Psycopg2 >= 2.5.4.

About the app:

- Available on pypi
- Tested using Travis CI
- Documentation on readthedocs.org
- Bug tracker

CHAPTER 1

Table of contents

1.1 Getting started

- Add `'modeltrans'` your list of `INSTALLED_APPS`.
- Make sure the current data in your models is in the language defined in the `LANGUAGE_CODE` django setting.
- By default, django-modeltrans uses the languages in the `LANGUAGES` django setting. If you want the list to be different, add a list of available languages to your `settings.py`: `MODELTRANS_AVAILABLE_LANGUAGES = ('en', 'nl', 'de', 'fr')`.
- Add a `modeltrans.fields.TranslationField` to your models and specify the fields you want to translate:

```
# models.py
from django.db import models
from modeltrans.fields import TranslationField


class Blog(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField(null=True)

    i18n = TranslationField(fields=("title", "body"))
```

- Run `./manage.py makemigrations` to add the `i18n` JSONField to each model containing translations.
- Each Model now has some extra virtual fields. In the example above:
  - `title` allows getting/setting the default language
  - `title_nl`, `title_de`, ... allow getting/setting the specific languages
  - If `LANGUAGE_CODE == 'en'`, `title_en` is mapped to `title`.
  - `title_i18n` follows the currently active translation in Django, and falls back to the default language

The above could be used in a Django shell like this:

```
>>> b = Blog.objects.create(title="Falcon", title_nl="Valk")
>>> b.title
'Falcon'
>>> b.title_nl
'Valk'
>>> b.title_i18n
'Falcon'
>>> from django.utils.translation import override
>>> with override('nl'):
...   b.title_i18n
...
'Valk'
# translations are stored in the field ``i18n`` in each model:
>>> b.i18n
{'title_nl': 'Valk'}
# if a translation is not available, None is returned.
>>> print(b.title_de)
None
# fallback to the default language
>>> with override("de"):
...     b.title_i18n
'Falcon'
# now, if we set the German translation, it it is returned from ``title_i18n``:
>>> b.title_de = 'Falk'
>>> with override('de'):
...     b.title_i18n
'Falk'
```

1.2 Forms

`TranslationModelForm` is an adaptation of Django's `django.forms.ModelForm` that allows management of translation fields. Assuming your model is translated with modeltrans, you can use `TranslationModelForm` to specify which languages to include form fields for.

For example, given a `NewsRoom` model:

```
class NewsRoom(models.Model):
    name = models.CharField(max_length=255)
    text = models.CharField(max_length=255)
    default_language = models.CharField(max_length=2)

    i18n = TranslationField(fields=("name", "text"), fallback_language_field="default_
↪language")
```

You can define a form using `TranslationModelForm` as:

```
from modeltrans.forms import TranslationModelForm

class NewsRoomTranslationForm(TranslationModelForm):

    class Meta:
        fields = ("name", "text")
        languages = ["browser", "fr", "fallback"]
        fallback_language = "en"
```

This defines a form with at most three language inputs per field, say `"nl"`, `"fr"` and `"en"`, where `"nl"` is the active browser language, and `"en"` the defined fallback language. `Meta.exclude` can also be used to define which fields are in the form, where the forms' `field_order` parameter can be used to define the field ordering.

1.2.1 Setting the form languages

`languages` defines the languages included in the form.

- **Options are:**
  - `"browser"`: the language that is active in the browser session
  - `"fallback"`: the fallback language either defined in the form, the model instance, or in the system, in that order of priority
  - a language code: e.g. `"fr"`, `"it"`
- Default: `["browser", "fallback"]`
- Ordering: the ordering defined in the declaration is preserved
- Duplicate languages are removed, e.g. `["browser", "fr", "fallback"]`, becomes `["fr"]` if browser language and fallback are also `"fr"`.

`languages` can be defined in the form `Meta` options as in the example above, or as a form kwarg as in:

```
form = NewsRoomTranslationForm(languages=["it", "fallback"])
```

1.2.2 Setting the fallback language

`fallback_language` defines the fallback language in the form. Requires `"fallback"` to be included in `languages`. Can be defined via the form `Meta` options as in the example above, and also be passed as a kwarg like `languages`. The following prioritization is followed:

1. **`fallback_language` passed as form parameter:** `Form(fallback_language="fr")`
2. **the `Meta` option `fallback_language`:** e.g. `class Meta: fallback_language = "fr"`
3. **A custom fallback of a model instance set via `fallback_language_field`:** e.g. `i18n = TranslationField(fields=("title", "header"), fallback_language_field="language_code")`
4. The default language of the system: If no `Meta` option is given fallback reverts to `get_default_language()`

1.2.3 Handling of field properties

Properties of translation form fields are inherited from the form field that is generated for the original model field. The label of the field is adjusted to include the relevant language and to designate the field as a translation or default fallback field, as follows:

- translation fields: "field name (NL, translation language)"
- fallback field: "field name (EN, default language)"

1.3 Admin

By default, each field is displayed for each language configured for django-modeltrans. This might work for a couple of languages, but with 2 translated fields and 10 languages, it already is a bit unwieldy.

The `ActiveLanguageMixin` is provided to show only the default language (`settings.LANGUAGE_CODE`) and the currently active language. Use like this:

```
from django.contrib import admin
from modeltrans.admin import ActiveLanguageMixin

from .models import Blog


@admin.register(Blog)
class BlogAdmin(ActiveLanguageMixin, admin.ModelAdmin):
    pass
```

1.4 Advanced usage

1.4.1 Custom fallback language

By default, fallback is centrally configured with *MODELTRANS_FALLBACK*. That might not be sufficient, for example if part of the content is created for a single language which is not `LANGUAGE_CODE`.

In that case, it can be configured per-record using the `fallback_language_field` argument to `TranslationField`:

```
class NewsRoom(models.Model):
    name = models.CharField(max_length=255)
    default_language = models.CharField(max_length=2)

    i18n = TranslationField(fields=("name",), fallback_language_field="default_
↪language")
```

You can traverse foreign key relations too:

```
class Article(models.Model):
    content = models.CharField(max_length=255)
    newsroom = models.ForeignKey(NewsRoom)

    i18n = TranslationField(fields=("content",), fallback_language_field="newsroom__
↪default_language")
```

Note that

- if in this example no `newsroom` is set yet, the centrally configured fallback is used.
- the original field _always_ contains the language as configured by `LANGUAGE_CODE`.

With the models above:

```
nos = NewsRoom.objects.create(name="NOS (en)", default_language="nl", name_nl="NOS␣
↪(nl)")
article = Article.objects.create(
    newsroom=nos,
```

```
    content="US-European ocean monitoring satellite launches into orbit",
    content_nl="VS-Europeese oceaanbewakingssatelliet gelanceerd"
)

with override('de'):
    # If language 'de' is not available, the records default_language will be used.
    print(nos.name)  # 'NOS (nl)'

    #  If language 'de' is not available, the newsroom.default_language will be used.
    print(article.content)  # 'VS-Europeese oceaanbewakingssatelliet gelanceerd'
```

1.4.2 Inheritance of models with translated fields.

When working with model inheritance, you might want to have different parameters to the `i18n`-field for the parent and the child model. These parameters can be overridden using the `i18n_field_params` attribute and on the child class:

```
from django.db import models
from modeltrans.fields import TranslationField

class ParentModel(models.Model):
    info = models.CharField(max_length=255)

    i18n = TranslationField(fields=("info",), required_languages=("en",))


class ChildModel(ParentModel):
    child_info = models.CharField(max_length=255)

    i18n_field_params = {
        "fields": ("info", "child_info"),
        "required_languages": ("nl",)
    }
```

1.5 Database performance

1.5.1 Adding GIN indexes

In order to perform well while filtering or ordering on translated values, the `i18n`-field need a GIN index. Due to limitations in the way Django currently allows to define indexes, they should be added manually:

```
from django.contrib.postgres.indexes import GinIndex
from django.db import models


class Category(models.Model):
    name = models.CharField(max_length=255)

    i18n = TranslationField(fields=("name",))

    class Meta:
        indexes = [GinIndex(fields=["i18n"]), ]
```

1.6 Inner workings

Django-modeltrans uses a `django.contrib.postgres.JSONField` to store field translations in the table and adds some augmentation to the queries made by Django's QuerySet methods to allow transparent use of the translated values.

The inner workings are illustrated using this model:

```
class Blog(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField(null=True)

    i18n = TranslationField(fields=("title", "body"))
```

When creating an object, translated fields in the constructor are transformed into a value in the i18n field. So the following two calls are equivalent:

```
Blog.objects.create(title="Falcon", title_nl="Valk", title_de="Falk")
Blog.objects.create(title="Falcon", i18n={"title_nl": "Valk", "title_de": "Falk"})
```

So adding a translated field does not need any migrations: it just requires adding a key to the `i18n` field.

When selecting objects django-modeltrans replaces any occurrence of a translated field with the appropriate JSONB key get operation:

```
Blog.objects.filter(title_nl="Valk")
# SELECT ... FROM "app_blog" WHERE (app_blog.i18n->>'title_nl')::varchar(255) = 'Valk'

Blog.objects.filter(title_nl__contains="a")
# SELECT ... FROM "app_blog" WHERE (app_blog.i18n->>'title_nl')::varchar(255) LIKE '%a
↪%'
```

In addition to that, you can use `<fieldname>_i18n` to filter on. That will use `COALESCE` to look in both the currently active language and the default language:

```
from django.utils.translation import override

with override("nl"):
    Blog.objects.filter(title_i18n="Valk")

# SELECT ... FROM "app_blog"
# WHERE COALESCE((app_blog.i18n->>'title_nl'), "app_blog"."title") = 'Valk'
```

Model objects containing translated fields get virtual fields for each field/ language combination plus a field which always returns the active language. In the example, we have configured 3 translation languages: `('nl', 'de', 'fr')` resulting in 4 virtual fields for each original field:

```
b = Blog.objects.create(title='Falcon', title_nl='Valk', title_de='Falk')
b._meta.get_fields()

(<django.db.models.fields.AutoField: id>,
 <django.db.models.fields.CharField: title>,
 <django.db.models.fields.TextField: body>,
 <django.db.models.fields.related.ForeignKey: category>,
 <modeltrans.fields.TranslationField: i18n>,
 <modeltrans.fields.TranslatedCharField: title_i18n>,
 <modeltrans.fields.TranslatedCharField: title_en>,
 <modeltrans.fields.TranslatedCharField: title_nl>,
```

```
 <modeltrans.fields.TranslatedCharField: title_de>,
 <modeltrans.fields.TranslatedCharField: title_fr>,
 <modeltrans.fields.TranslatedTextField: body_i18n>,
 <modeltrans.fields.TranslatedTextField: body_en>,
 <modeltrans.fields.TranslatedTextField: body_nl>,
 <modeltrans.fields.TranslatedTextField: body_de>,
 <modeltrans.fields.TranslatedTextField: body_fr>)
```

Each virtual field for an explicit language will only return a value if that language is defined:

```
print(b.title_nl, b.title_fr)
# 'Valk', None
```

The virtual field `<field>_i18n` returns the translated value for the current active language and falls back to the language in `LANGUAGE_CODE`:

```
with override("nl"):
    print(b.title_i18n)
# 'Valk'

with override("de"):
    print(b.title_i18n)
# 'Falk'

with override("fr"):
    print(b.title_i18n)
# 'Falcon' (no french translation available, falls back to LANGUAGE_CODE)
```

Django-modeltrans also allows ordering on translated values. Ordering on `<field>_i18n` probably makes most sense, as it more likely that there is a value to order by:

```
with override("de"):
    qs = Blog.objects.order_by("title_i18n")

# SELECT ...,
# FROM "app_blog"
# ORDER BY COALESCE((app_blog.i18n->>'title_de'), "app_blog"."title") ASC
```

Results in the following ordering:

```
title_i18n   title_en     title_nl     title_de
------------ ------------ ------------ ------------
Crayfish     Crayfish
Delfine      Dolphin      Dolfijn      Delfine
Dragonfly    Dragonfly    Libellen
Duck         Duck         Eend
Falk         Falcon       Valk         Falk
Frog         Frog         Kikker
Kabeljau     Cod                       Kabeljau
Toad         Toad         Pad
```

As you can see, although the german translations are not complete, ordering on `title_i18n` still results in a useful ordering.

Note: These examples assume the default setting for `MODELTRANS_FALLBACK`. If you customize that setting, it can get slightly more complex, resulting in more than 2 arguments to the `COALESCE` function.

1.7 Known issues

We use django-modeltrans in production, but some aspects of it's API might be a bit surprising. This page lists the issue we are aware of. Some might get fixed at some point, some are just the result of database or Django implementations.

Reading the explanation of the *Inner workings* might also help to understand some of these issues.

1.7.1 Unsupported QuerySet methods

Using translated fields in `QuerySet`/`Manager` methods `.distinct()`, `.extra()`, `.aggregate()`, `.update()` is not supported.

1.7.2 Fields supported

Behavior is tested using `CharField()` and `TextField()`, as these make most sense for translated values. Additional fields could make sense, and will likely work, but need extra test coverage.

1.7.3 Ordering defined in `Model.Meta.ordering`

Any ordering using translated fields defined in `Model.Meta.ordering` is only supported with Django 2.0 and later (django/django#8473 is required).

1.7.4 Context of 'current language'

Lookups (`<field>_i18n`) are translated when the line they are defined on is executed:

```
class Foo():
    qs = Blog.objects.filter(title_i18n__contains="foo")

    def get_blogs(self):
        return self.qs
```

When `Foo.get_blogs()` will be called in the request cycle, one might expect the current language for that request to define the `title_i18n__contains` filter. But instead, the language active while creating the class `Foo` will be used.

For example the `queryset` argument to `ModelChoiceField()`. See github issue #34

1.7.5 Using translated fields from a related model

`modeltrans.manager.MultilingualManager` is required to transform `<field>_i18n` to the correct lookup in the `JSONField`.

If you want to use translated fields when building the query from a related model, you need to add `objects = MultilingualManager()` to the model you want to build the query from.

For example, `Category` needs `objects = MultilingualManager()` in order to allow `Category.objects.filter(blog__title_i18n__icontains="django")`:

```
class Category(models.Model):
    title = models.CharField(max_length=255)

    objects = MultilingualManager()  # required to use translated fields of Blog.

class Blog(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField(null=True)
    category = models.ForeignKey(Category, null=True, blank=True, on_delete=models.
↪CASCADE)

    i18n = TranslationField(fields=("title", "body"))
```

1.8 Related packages

1.8.1 django-modeltranslation

Uses one field for each language/field combination, so having 3 languages and 3 translatable fields will result in 9 extra fields on each database table. It rewrites queries in order to return the current language, but the contents of the original field are undetermined if a field is translated.

1.8.2 django-nece

Also uses a `jsonb` PostgreSQL field, but has a bunch of custom `QuerySet` and `Model` methods to get translated values. It also requires one to inherit from a `TranslationModel`.

1.8.3 django-i18nfield

Stores JSON in a `TextField`, so does not allow lookup, searching or ordering by the translated fields.

1.9 Management commands

The packages adds a management command to create relevant migrations not automatically created by the `./manage.py makemigrations` command.

1.9.1 Data migration to migrate from django-modeltranslation

Syntax: `./manage.py i18n_makemigrations <apps>`

Only to migrate data from the fields managed by django-modeltranslation to the JSON field managed by django-modeltrans.

Explained in more detail in *Migrating from django-modeltranslation*

1.10 API Reference

1.10.1 Public API

modeltrans.admin

class modeltrans.admin.**ActiveLanguageMixin**
Add this mixin to your admin class to exclude all virtual fields, except:

- The original field (for the default language, settings.LANGUAGE_CODE)
- The field for the currently active language, without fallback.

modeltrans.apps

class modeltrans.apps.**RegistrationConfig**(*app_name*, *app_module*)

modeltrans.fields

class modeltrans.fields.**TranslatedVirtualField**(*original_field*, *language=None*, **args*, ***kwargs*)
A field representing a single field translated to a specific language.

Parameters

- **original_field** – The original field to be translated
- **language** – The language to translate to, or `None` to track the current active Django language.

get_field_name()
Returns the field name for the current virtual field.

The field name is `<original_field_name>_<language>` in case of a specific translation or `<original_field_name>_i18n` for the currently active language.

get_language()
Returns the language for this field.

In case of an explicit language (title_en), it returns "en", in case of `title_i18n`, it returns the currently active Django language.

class modeltrans.fields.**TranslationField**(*fields=None*, *required_languages=None*, *virtual_fields=True*, *fallback_language_field=None*, **args*, ***kwargs*)
This model field is used to store the translations in the translated model.

Parameters

- **fields** (*iterable*) – List of model field names to make translatable.
- **required_languages** (*iterable or dict*) – List of languages required for the model. If a dict is supplied, the keys must be translated field names with the value containing a list of required languages for that specific field.
- **virtual_fields** (*bool*) – If `False`, do not add virtual fields to access translated values with. Set to `True` during migration from django-modeltranslation to prevent collisions with it's database fields while having the `i18n` field available.

- **fallback_language_field** – If not None, this should be the name of the field containing a language code to use as the first language in any fallback chain. For example: if you have a model instance with 'nl' as language_code, and set fallback_language_field='language_code', 'nl' will always be tried after the current language before any other language.

modeltrans.manager

class modeltrans.manager.**MultilingualManager**

When adding the *modeltrans.fields.TranslationField* to a model, MultilingualManager is automatically mixed in to the manager class of that model.

If you want to use translated fields when building the query from a related model, you need to add `objects = MultilingualManager()` to the model you want to build the query from.

For example, `Category` needs `objects = MultilingualManager()` in order to allow `Category.objects.filter(blog__title_i18n__icontains="django")`:

```
class Category(models.Model):
    title = models.CharField(max_length=255)

    objects = MultilingualManager()  # required to use translated fields of Blog.

class Blog(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField(null=True)
    category = models.ForeignKey(Category, null=True, blank=True, on_
↪delete=models.CASCADE)

    i18n = TranslationField(fields=("title", "body"))
```

class modeltrans.manager.**MultilingualQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Extends `~django.db.models.query.QuerySet` and makes the translated versions of fields accessible through the normal QuerySet methods, analogous to the virtual fields added to a translated model:

- `<field>` allow getting/setting the default language
- `<field>_<lang>` (for example, `<field>_de`) allows getting/setting a specific language. Note that if `LANGUAGE_CODE == "en"`, `<field>_en` is mapped to `<field>`.
- `<field>_i18n` follows the currently active translation in Django, and falls back to the default language.

When adding the *modeltrans.fields.TranslationField* to a model, MultilingualManager is automatically mixed in to the manager class of that model.

1.11 Settings Reference

django-modeltrans allows some configuration to define its behavior. By default, it tries to use sensible defaults derived from the default django settings.

1.11.1 MODELTRANS_AVAILABLE_LANGUAGES

A list of language codes to allow model fields to be translated in. By default, the language codes extracted from django's LANGUAGES setting.

Note that

- the default language, defined in django's LANGUAGE_CODE setting, should not be added to this list (will be ignored).
- order is not important

A custom definition might be:

```
MODELTRANS_AVAILABLE_LANGUAGES = ('de', 'fr')
```

1.11.2 MODELTRANS_FALLBACK

A dict of fallback chains as lists of languages. By default, it falls back to the language defined in django setting `LANGUAGE_CODE`.

For example, django-modeltrans will fall back to:

- english when the active language is 'nl'
- fist dutch and finally english with active language is 'fy'

If configured like this:

```
LANGUAGE_CODE = 'en'
MODELTRANS_AVAILABLE_LANGUAGES = ('nl', 'fy')
MODELTRANS_FALLBACK = {
   'default': (LANGUAGE_CODE, ),
   'fy': ('nl', 'en')
}
```

Note that a custom fallback language can be configured on a model instance if the `i18n` field is configured like this:

```
class Model(models.Model):
    title = models.CharField(max_length=100)
    fallback_language = models.CharField(max_length=2)

    i18n = TranslationField(fields=("title",), fallback_language_field="fallback_
↪language")
```

in which `fallback_language_field` refers to the model field that contains the language code.

This topic is explained in *Custom fallback language*.

1.11.3 MODELTRANS_ADD_FIELD_HELP_TEXT

If `True`, the `<name>_i18n` fields with empty `help_text``s will get a ``help_text` like:

```
current language: en
```

`True` by default.

1.12 Migrating from django-modeltranslation

This is how to migrate from django-modeltranslation (version 0.12.1) to `django-modeltrans`:

1. Make sure you have a recent backup of your data available!

2. Add `modeltrans` to your `INSTALLED_APPS`
3. Make sure the default language for django-modeltranslation is equal to the language in `LANGUAGE_CODE`, which django-modeltrans will use.
4. Copy the setting `AVAILABLE_LANGUAGES` to `MODELTRANS_AVAILABLE_LANGUAGES`.
5. Add the TranslationField to the models you want to translate and keep the registrations for now. In order to prevent field name collisions, disable the virtual fields in django-modeltrans for now (`virtual_fields=False`):

```
# models.py
from django.contrib.postgres.indexes import GinIndex
from django.db import models

from modeltrans.fields import TranslationField

class Blog(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField(null=True)

    # add this field, containing the TranslationOptions attributes as arguments:
    i18n = TranslationField(fields=('title', 'body'), virtual_fields=False)

    # add the GinIndex
    class Meta:
        indexes = [GinIndex(fields=["i18n"])]


# translation.py
from modeltranslation.translator import translator, TranslationOptions
from .models import Blog

class BlogTranslationOptions(TranslationOptions):
    fields = ('name', 'title', )

translator.register(Blog, BlogTranslationOptions)
```

6. Run `./manage.py makemigrations <apps>`. This will create the migration adding the `i18n`-fields required by django-modeltrans. Apply them with `./manage.py migrate`
7. We need to create a migration to copy the values of the translated fields into the newly created `i18n`-field. django-modeltrans provides a management command to do that `./manage.py i18n_makemigrations <apps>`.
8. Run `./manage.py migrate` to apply the generated data migrations. Your models with translated fields should have a populated `i18n` field after these migrations.
9. Now, to remove django-modeltranslation:
    - Remove `modeltranslation` from `INSTALLED_APPS`.
    - Remove django-modeltranslation settings (`DEFAULT_LANGUAGE`, `AVAILABLE_LANGUAGES`) from your `settings.py`'s
    - Remove all `translation.py` files from your apps.
    - Remove the use of `modeltranslation.admin.TranslationAdmin` in your `admin.py`'s
10. Run `./manage.py makemigrations <apps>`. This will create migrations that remove the fields generated by django-modeltranslation from your registered models.

11. Run `./manage.py migrate` to actually apply the generated migrations. This will remove the django-modeltranslation fields and their content from your database.
12. Update your code:
    - Remove `virtual_fields=False` from each `TranslationField`.
    - Use the `<field>_i18n` field in places where you would use `<field>` with django-modeltranslation. Less magic, but explicit is better than implicit!
    - Use `<field>_<language>` for the translated fields, just like your are used to.
    - If you use lookups containing translated fields from non-translated models, you should add `MultilingualManager()` to your models as a manager:

```
from django.contrib.postgres.indexes import GinIndex
from django.db import models

from modeltrans.fields TranslationField
from modeltrans.manager import MultilingualManager


class Site(models.Model):
    title = models.CharField(max_length=100)

    # adding manager allows queries like Site.objects.filter(blog__title_i18n_
↪_contains='modeltrans')
    objects = MultilingualManager()


class Blog(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()

    i18n = TranslationField(fields=('title', 'body'))
    site = models.ForeignKey(Site)

    class Meta:
        indexes = [GinIndex(fields=["i18n"]]
```

1.13 django-modeltrans change log

1.13.1 0.7.5 (2023-08-03)

- Fix crash with `limit_choices_to` (#100) fixes: #94

1.13.2 0.7.4 (2023-04-03)

- Add support for Django 4.2

1.13.3 0.7.3 (2022-05-25)

- Revert: Use `TranslationModelForm` to implement `ActiveLanguageMixin` (#86)

1.13.4 0.7.2 (2021-10-21)

- Use `TranslationModelForm` to implement `ActiveLanguageMixin` (#80)
- Add support for django 4.0 (#78)

1.13.5 0.7.1 (2021-07-21)

- Add translations for NL, DE and FR
- Remove translation strings in example and tests

1.13.6 0.7.0 (2021-06-30)

- Add `TranslationModelForm`

1.13.7 0.6.0 (2021-04-07)

- Add support for Django==3.2

1.13.8 0.5.2 (and 0.5.1) (2021-01-12)

- Adjust imports to remove deprecation warnings with django==3.1.* #65

1.13.9 0.5.0 (2020-11-23)

- Add per-record fallback feature #63

1.13.10 0.4.0 (2019-11-22)

- Drop python 2, Django 1.11 and Django 2.0 support #56
- Add option for `i18n` model fields inheritance #51
- Fix patching managers of models inheriting from an abstract models #50

1.13.11 0.3.4 (2019-01-15)

- Fix exception on nullable i18n field #49

1.13.12 0.3.3 (2018-07-19)

- Add instruction to remove `virtual_fields=True` to migration guide, fixes #45
- Use `AppConfig` to compute path to app dir, fixes #46
- Do not copy empty fields into i18n field, fixes #47

1.13.13 0.3.2 (2018-07-18)

- Removed `encoding` kwarg to `open()` in `setup.py` to fix python 2.7 install.

1.13.14 0.3.1 (2018-07-16)

- Added `long_description` to `setup.py`, no functional changes.

1.13.15 0.3.0 (2018-07-15)

- Adopted black code style.
- Removed auto-adding indexes, as it was unpredictable. You must add the `GinIndex` manually like described in the documentation on performance.
- Support dict for `required_languages` argument to `TranslationField`, to allow more fine-grained mapping of field names to required languages.
- `ActiveLanguageMixin` does not use the `<field>_i18n` version of the field, but rather the virtual field with the current active language. This makes sure no fallback values are accidentally saved for another language.

1.13.16 0.2.2 (2018-03-13)

- Hide original field with `ActiveLanguageMixin`.
- Raise an `ValueError` on accessing translated fields on a model fetched with `.defer('i18n')`.
- do not accidentally add `i18n` to **dict** in Model.create
- Improve handling of explicit PK's and expression rewriting code.
- Add help_text to virtual fields with language=None.

1.13.17 0.2.1 (2018-01-24)

- Dropped support for Django 1.9 and 1.10.
- Used `ugettext_lazy` rather than `ugettext` to fix admin header translation #32
- Removed default value `False` for `Field.editable`, to allow using the translated version of a field in a `ModelForm`.

1.13.18 0.2.0 (2017-11-13)

- No annotations are made while ordering anymore, instead, expressions are passed onto the original `order_by()` method.
- Any translated fields used in `Model.Meta.ordering` is transformed into the correct expression with django 2.0 and later (fixes #25).
- `django.contrib.postgres.GinIndex` is added to the `i18n` column if it's supported by the django version used (1.11 and later). It can be disabled with the setting `MODELTRANS_CREATE_GIN`.
- The migration generated from `./manage.py i18n_makemigrations <app>` used to move the data and add a GIN index. This is split into two commands: `./manage.py i18n_makemigrations` and `./manage.py i18n_make_indexes`.

- Added support for ‘values(**expressions)‘‘ with references to translated fields.
- Added support for translated values in `annotate()`

1.13.19 0.1.2 (2017-10-23)

- Ensure a dynamic mixed `MultilingualQuerySet` can be pickled.
- Add basic support for `Func` in `order_by()`

1.13.20 0.1.1 (2017-10-23)

- Allow adding `MultilingualManager()` as a manager to objects without translations to allow lookups of translated content through those managers.

1.13.21 0.1.0 (2017-10-23)

- Use proper alias in subqueries, fixes #23.
- Support lookups on and ordering by related translated fields (`.filter(category__name_nl='Vogels')`), fixes #13.
- Use `KeyTextTransform()` rather than `RawSQL()` to access keys in the `JSONField`. For Django 1.9 and 1.10 the Django 1.11 version is used.

1.13.22 0.0.8 (2017-10-19)

- Check if `MODELTRANS_AVAILABLE_LANGUAGES` only contains strings.
- Make sure `settings.LANGUAGE_CODE` is never returned from `conf.get_available_languages()`

1.13.23 0.0.7 (2017-09-04)

- Cleaned up the settings used by django-modeltrans #19. This might be a breaking change, depending on your configuration.
  - `AVAILABLE_LANGUAGES` is now renamed to `MODELTRANS_AVAILABLE_LANGUAGES` and defaults to the language codes in the django `LANGUAGES` setting.
  - `DEFAULT_LANGUAGE` is removed, instead, django-modeltrans uses the django `LANGUAGE_CODE` setting.
- Added per-language configurable fallback using the `MODELTRANS_FALLBACK` setting.

1.13.24 0.0.6 (2017-08-29)

- Also fall back to `DEFAULT_LANGUAGE` if the value for a key in the translations dict is falsy.

1.13.25 0.0.5 (2017-07-26)

- Removed registration in favor of adding the `TranslationField` to a model you need to translated.
- Created documentation.

1.13.26 0.0.4 (2017-05-19)

- Improve robustness of rewriting lookups in QuerySets

1.13.27 0.0.3 (2017-05-18)

- Add the `gin` index in the data migration.
- Added tests for the migration procedure.

Index

A

ActiveLanguageMixin (class in modeltrans.admin), 12

G

get_field_name() (modeltrans.fields.TranslatedVirtualField method), 12
get_language() (modeltrans.fields.TranslatedVirtualField method), 12

M

MultilingualManager (class in modeltrans.manager), 13
MultilingualQuerySet (class in modeltrans.manager), 13

R

RegistrationConfig (class in modeltrans.apps), 12

T

TranslatedVirtualField (class in modeltrans.fields), 12
TranslationField (class in modeltrans.fields), 12