Wagtail Modeltranslation Docs Documentation

Release latest

July 06, 2016

Contents

1 Features **3**

2 Contents **5**

2.1 Introduction 5
2.2 Installation 6
2.3 Advanced Settings 7
2.4 Registering models for translation 11
2.5 Accessing translated fields 14
2.6 Multilingual Manager 16
2.7 Authors 20
2.8 Change Log 20

This app is based on django-modeltranslation: https://github.com/deschler/django-modeltranslation

It's an alternative approach for i18n support on Wagtail CMS websites.

The modeltranslation application is used to translate dynamic content of existing Wagtail models to an arbitrary number of languages, without having to change the original model classes. It uses a registration approach (comparable to Django's admin app) to add translations to existing or new projects and is fully integrated into the Wagtail admin UI.

The advantage of a registration approach is the ability to add translations to models on a per-app basis. You can use the same app in different projects, whether or not they use translations, and without touching the original model class.

CHAPTER 1

Features

- Add translations without changing existing models or views
- Translation fields are stored in the same table (no expensive joins)
- Supports inherited models (abstract and multi-table inheritance)
- Handle more than just text fields
- Wagtail admin integration
- Flexible fallbacks, auto-population and more!
- Default Page model fields has translatable fields by default
- StreamFields are now supported!

CHAPTER 2

Contents

2.1 Introduction

2.1.1 Creating multilingual sites

I18n

Django and Wagtail CMS have implemented Internationalisation (I18n) in their frameworks. Hooks are provided for translating strings such as literals. Furthermore, **locale language files** are included. This is where the translated text of the frameworks is stored.

When writing your own apps, it is recommended that you use Il18n. If you need guidance, you can read the Django Internalization Documentation.

Wagtail-modeltranslation

Another important component in the translation equation is the content stored in database fields. This is where wagtail-modeltranslation comes into play.

Wagtail-modeltranslation is a fork of django-modeltranslation designed to define the fields that need to be translated. In Wagtail, translation fields are displayed and edited together on the same page in the Wagtail admin interface. Translated fields can be used in your templates and as you would use any other field.

Some of the advantages of wagtail-modeltranslation

- The same template is used for multiple languages
- The document tree is simpler with no need to have a separate branch for each language
- Languages can be added without changing existing models or views
- Translation fields are stored in the same table (no expensive joins)
- Can handle more than just text fields
- Wagtail admin integration
- Flexible fallbacks, auto-population and more!
- Default Page model has translatable fields by default
- StreamFields are supported
- Easy to implement

Examples used in this document

We will be using a fictitious model `foo` in the coding examples.

Wagtail-modeltranslation and Modeltranslation

This document is for the most part an adaptation of the django-modeltranslation documentation, so we will refer to `wagtail-modeltranslation` when the material discussed is specific to Wagtail CMS and `modeltranslation` when it is applicable to both wagtail-modeltranslation and django-modeltranslaton. You don't need to distinguish between the two since wagtail-modeltranslation includes all the functionalities of django-modeltranslation.

2.2 Installation

2.2.1 Requirements

- Django >= 1.7
- Wagtail >= 1.0

Installing using Pip

```
$ pip install wagtail-modeltranslation >= 0.2.2
```

Installing using the source

- From github: **git clone** https://github.com/infoportugal/wagtail-modeltranslation.git
  - Copy wagtail_modeltranslation folder in project tree

  OR
- Download ZIP file on Github.com from infoportugal/wagtail-modeltranslation
  - Unzip and copy wagtail_modeltranslation folder in project tree

2.2.2 Setup

To setup the application please follow these steps:

1. In the settings/base.py file:
   - Add wagtail_modeltranslation to the INSTALLED_APPS

```
INSTALLED_APPS = (
 ...
'wagtail_modeltranslation',
 )
```

   - Add django.middleware.locale.LocaleMiddleware to MIDDLEWARE_CLASSES.

```
MIDDLEWARE_CLASSES = (
...

'django.middleware.locale.LocaleMiddleware',
)
```

- Set `USE_I18N = True`

- Configure your LANGUAGES.

  The LANGUAGES variable must contain all languages you will use for translation. The first language is treated as the *default language*.

  Modeltranslation uses the list of languages to add localized fields to the models registered for translation. For example, to use the languages Portuguese, Spanish and French in your project, set the LANGUAGES variable like this (where `pt` is the default language).

```
LANGUAGES = (
   ('pt', u'Portugese'),
   ('es', u'Spanish'),
   ('fr', u'French'),
 )
```

Warning: When the LANGUAGES setting isn't present in `settings/base.py` (and neither is `MODELTRANSLATION_LANGUAGES`), it defaults to Django's global LANGUAGES setting instead, and there are quite a few languages in the default!

2. Create a `translation.py` file in your app directory and register `TranslationOptions` for every model you want to translate.

```
from .models import foo
from wagtail_modeltranslation.translation import TranslationOptions
from wagtail_modeltranslation.decorators import register

@register(foo)
class FooTR(TranslationOptions):
    fields = (
       'body',
    )
```

3. Add `TranslationMixin` to your translatable model:.

```
# .models foo
...
from wagtail_modeltranslation.models import TranslationMixin

class FooModel(TranslationMixin, Page):
    body = StreamField(...)
```

4. Run `python manage.py makemigrations` followed by `python manage.py migrate`. This will add extra fields in the database.

2.3 Advanced Settings

Modeltranslation has some advanced settings to customize its behaviour.

2.3.1 Default language

`MODELTRANSLATION_DEFAULT_LANGUAGE`

Default: `None`

To override the default language as described in *Configuration settings*, you can define a language in `MODELTRANSLATION_DEFAULT_LANGUAGE`. Note that the value has to be in `settings.LANGUAGES`, otherwise an `ImproperlyConfigured` exception will be raised.

Example:

```
MODELTRANSLATION_DEFAULT_LANGUAGE = 'pt'
```

2.3.2 Default languages

`MODELTRANSLATION_LANGUAGES`

Default: same as LANGUAGES

Allows to set the languages the content will be translated into. If not set, by default all languages listed in LANGUAGES will be used.

Example:

```
LANGUAGES = (
    ('pt', 'Portuguese'),
    ('es', 'Spanish'),
    ('fr', 'French'),
)

MODELTRANSLATION_LANGUAGES = ('pt', 'es')
```

Note: This setting may become useful if your users will be producing content for a restricted set of languages, while your application is translated into a greater number of locales.

2.3.3 Fallback languages

`MODELTRANSLATION_FALLBACK_LANGUAGES`

Default: `(DEFAULT_LANGUAGE)`

By default modeltranslation will fallback to the computed value of the DEFAULT_LANGUAGE. This is either the first language found in the LANGUAGES setting or the value defined through MODELTRANSLATION_DEFAULT_LANGUAGE which acts as an override.

This setting allows for a more fine grained tuning of the fallback behaviour by taking additional languages into account. The language order is defined as a tuple or list of language codes.

Example:

```
MODELTRANSLATION_FALLBACK_LANGUAGES = ('pt', 'es')
```

Using a dict syntax it is also possible to define fallbacks by language. A default key is required in this case to define the default behaviour of unlisted languages.

Example:

```
MODELTRANSLATION_FALLBACK_LANGUAGES = {'default': ('pt', 'es'), 'fr': ('es',)}
```

Note: Each language has to be in the LANGUAGES setting, otherwise an `Improperly Configured` exception is raised.

2.3.4 Prepopulate language

`MODELTRANSLATION_PREPOPULATE_LANGUAGE`

Default: `the current active language`

By default modeltranslation will use the current request language for prepopulating admin fields specified in the `prepopulated_fields` admin property. This is often used to automatically fill slug fields.

This setting allows you to pin this functionality to a specific language.

Example:

```
MODELTRANSLATION_PREPOPULATE_LANGUAGE = 'fr'
```

Note: The language has to be in the `LANGUAGES` setting, otherwise an `ImproperlyConfigured` exception is raised.

2.3.5 Translation files

`MODELTRANSLATION_TRANSLATION_FILES`

Default: `()` (empty tuple)

Modeltranslation uses an autoregister feature similiar to the one in Django's admin. The autoregistration process will look for a `translation.py` file in the root directory of each application that is in INSTALLED_APPS.

The setting `MODELTRANSLATION_TRANSLATION_FILES` is provided to extend the modules that are taken into account.

Syntax:

```
MODELTRANSLATION_TRANSLATION_FILES = (
    '<APP1_MODULE>.translation',
    '<APP2_MODULE>.translation',
)
```

Example:

```
MODELTRANSLATION_TRANSLATION_FILES = (
    'news.translation',
    'projects.translation',
)
```

2.3.6 Custom fields

`MODELTRANSLATION_CUSTOM_FIELDS`

Default: `()` (empty tuple)

Modeltranslation supports the fields listed in the **'Matrix of supported_fields'_**. In most cases subclasses of the supported fields will work fine, too. Unsupported fields will throw an `Improperly Configured` exception.

The list of supported fields can be extended by defining a tuple of field names in your `settings file`.

Example:

```
MODELTRANSLATION_CUSTOM_FIELDS = ('MyField', 'MyOtherField',)
```

Warning: This just prevents modeltranslation from throwing an `Improperly Configured` exception. Any unsupported field will most likely fail in one way or another. The feature is considered experimental and might be replaced by a more sophisticated mechanism in future versions.

2.3.7 Auto populate

`MODELTRANSLATION_AUTO_POPULATE`

Default: `False`

This setting controls if the *multilingual_manager* should automatically populate language field values in its `create` and `get_or_create` method, and in model constructors, so that these two blocks of statements can be considered equivalent:

```
foo.objects.populate(True).create(title='-- no translation yet --')
with auto_populate(True):
    q = foo(title='-- no translation yet --')

# same effect with MODELTRANSLATION_AUTO_POPULATE == True:

foo.objects.create(title='-- no translation yet --')
q = foo(title='-- no translation yet --')
```

2.3.8 Debug

`MODELTRANSLATION_DEBUG`

Default: `False`

Used for modeltranslation related debug output. Currently setting it to `False` will just prevent Django's development server from printing the `Registered xx models for translation` message to stdout.

2.3.9 Fallbacks

`MODELTRANSLATION_ENABLE_FALLBACKS`

Default: `True`

Controls if fallback (both language and value) will occur.

2.3.10 Retain locale

`MODELTRANSLATION_LOADDATA_RETAIN_LOCALE`

Default: `True`

Control if the `loaddata` command should leave the settings-defined locale alone. Setting it to `False` will result in previous behaviour of loaddata: inserting fixtures to database under en-us locale.

2.4 Registering models for translation

`Modeltranslation` can translate model fields of any model class.

In wagtail-modeltranslation a **TranslationMixin** is used with the Page model:

Registering models and their fields used for translation requires the following steps:

1. Create **translation.py** in your app directory.
2. Define the models you want to use, import wagtail-modeltranslation's **TranslationOptions** and the **register** decorator
3. Create a translation option class for every model you want to translate and precede the class with the **@register** decorator.

The wagtail-modeltranslation application reads the **translation.py** file in your app directory thereby triggering the registration of the translation options found in the file.

A translation option is a class that declares which model fields are needed for translation. The class must derive from **wagtail_modeltranslation.translator.TranslationOptions** and it must provide a **field** attribute storing the list of field names. The option class must be registered with the **wagtail_modeltranslation.decorators.register** instance.

To illustrate this let's have a look at a simple example using a **Foo** model. The example only contains an **introduction** and a **body** field.

Instead of a **Foo** model, this could be any Wagtail model class:

```
from .models import Foo
from wagtail_modeltranslation.translation import TranslationOptions
from wagtail_modeltranslation.decorators import register

@register(Foo)
class FooTR(TranslationOptions):
    fields = (
        'introduction',
        'body',
    )
```

In the above example, the **introduction** and **body** language fields will be be added for each language defined in LANGUAGES in the settings file ,**base.py**, when the database is updated with **./manage.py makemigrations** and **./manage.py migrate.**

At this point you are mostly done and the model classes registered for translation will have been added some auto-magical fields. The next section explains how things are working under the hood.

2.4.1 Changes automatically applied to the model class

After registering the **Foo** model for translation a SQL dump of the Foo app will look like this:

```
$ ./manage.py sqlall news
BEGIN;
CREATE TABLE `news_Foo` (
    `id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `introduction` varchar(255) NOT NULL,
```

```
    `introduction_pt` varchar(255) NULL,
    `introduction_es` varchar(255) NULL,
    `introduction_fr` varchar(255) NULL,
    `body` varchar(255) NOT NULL,
    `body_pt` varchar(255) NULL,
    `body_es` varchar(255) NULL,
    `body_fr` varchar(255) NULL,
)
;
CREATE INDEX `news_Foo_page_id` ON `news_Foo` (`page_id`);
COMMIT;
```

Note the `introduction_pt`, `introduction_es`, `introduction_fr`, `body_pt`, `body_es` and `body_fr` fields which are not declared in the original `Foo` model class have been added by the modeltranslation app. These are called *translation fields*. There will be one for every language in your project's `settings.py`.

The name of these additional fields is build using the original name of the translated field and appending one of the language identifiers found in the `settings.LANGUAGES`.

As these fields are added to the registered model class as fully valid Django model fields, they will appear in the db schema for the model although it has not been specified on the model explicitly.

2.4.2 Precautions regarding registration approach

Be aware that registration approach (as opposed to base-class approach) to models translation has a few caveats, though (despite many pros).

First important thing to note is the fact that translatable models are being patched - that means their fields list is not final until the modeltranslation code executes. In normal circumstances it shouldn't affect anything - as long as `models.py` contain only models' related code.

For example: consider a project where a `ModelForm` is declared in `models.py` just after its model. When the file is executed, the form gets prepared - but it will be frozen with old fields list (without translation fields). That's because the `ModelForm` will be created before modeltranslation would add new fields to the model (`ModelForm` gathers fields info at class creation time, not instantiation time). Proper solution is to define the form in `forms.py`, which wouldn't be imported alongside with **models.py** (and rather imported from views file or urlconf).

Generally, for seamless integration with modeltranslation (and as sensible design anyway), the models.py‘‘ should contain only bare models and model related logic.

2.4.3 Committing fields to database

Modeltranslation supports the migration system introduced by Django 1.7. Besides the normal workflow as described in Django's Migration Docs, you should do a migration whenever one of the following changes have been made to your project:

- Added or removed a language through `settings.LANGUAGES` or `settings.MODELTRANSLATION LANGUAGES`.
- Registered or unregistered a field through `TranslationOptions`.

It doesn't matter if you are starting a fresh project or change an existing one, it's always:

1. `python manage.py makemigration` to create a new migration with the added or removed fields.
2. `python manage.py migrate` to apply the changes.

2.4.4 Required fields

By default, all translation fields are optional (not required). This can be changed using a special attribute on `TranslationOptions`:

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('introduction', 'body',)
    required_languages = ('pt', 'es')
```

It's quite self-explanatory: for Portuguese and Spanish, the `introduction` and `body` translation fields are required. For other languages, they are optional.

A more fine-grained control is available:

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('introduction', 'body',)
    required_languages = {'pt': ('introduction', 'body'), 'default': ('introduction',)}
```

For Portuguese, all fields (both `introduction` and `body`) are required; for all other languages, only `introduction` is required. The `default` is optional.

Note: Requirement is enforced by `blank=False`. Please remember that it will trigger validation only in modelforms and admin (as always in Django). Manual model validation can be performed via the `full_clean()` model method.

The required fields are still `null=True`, though.

2.4.5 Matrix of supported fields

While the main purpose of modeltranslation is to translate text-like fields, translating other fields can be useful in several situations. The table lists all model fields available in Django and Wagtail and gives an overview about their current support status.

Model Field	Implemented
AutoField	No
BigIntegerField	Yes*
BooleanField	Yes
CharField	Yes
CommaSeparatedIntegerField	Yes
DateField	Yes
DateTimeField	Yes
DecimalField	Yes
EmailField	Yes*
FileField	Yes
FilePathField	Yes*
FloatField	Yes
ImageField	Yes
IntegerField	Yes
IPAddressField	Yes
GenericIPAddressField	Yes
NullBooleanField	Yes
PositiveIntegerField	Yes*
PositiveSmallIntegerField	Yes*
SlugField	Yes*
SmallIntegerField	Yes*
StreamField	Yes
TextField	Yes
TimeField	Yes
URLField	Yes*
ForeignKey	Yes
OneToOneField	Yes
ManyToManyField	No

* Implicitly supported (as subclass of a supported field)

2.5 Accessing translated fields

Modeltranslation changes the behaviour of the translated fields. To explain this consider the Foo example from the *Registering models for translation* chapter again. The original `Foo` model:

```
class FooModel(TranslationMixin, Page):
    introduction = models.CharField(max_length=255)
    body = RichtextField(blank=True)
```

Now that it is registered with wagtail-modeltranslation, the model looks like this - note the additional fields automatically added by the app:

```
class FooModel(TranslationMixin, Page):
   introduction = models.CharField(max_length=255)  # original/translated field
   introduction_pt = models.CharField(null=True, blank=True, max_length=255)  # default translation f
   introduction_es = models.CharField(null=True, blank=True, max_length=255)  # translation field
   introduction_fr = models.CharField(null=True, blank=True, max_length=255)  # translation field
   body = RichTextField(blank=True)  # original/translated field
   body_pt = RichTextField(null=True, blank=True)  # default translation field
   body_es = RichTextField(null=True, blank=True)  # translation field
   body_fr = RichTextField(null=True, blank=True)  # translation field
```

The example above assumes that the default language is `pt`, therefore the `introduction_pt` and `body_pt` fields are marked as the *default translation fields*. If the default language were `fr`, then `introduction_fr` and `body_fr` would be the *default translation fields*.

Warning: The `title` field is inherited from the Page model; if you try to create a field called `title` in one of your `models` you will get a warning messaqe. You can include the `title field` in `translation.py` and in the `content_panels` since it is inherited.

```
# Indicate fields to include in Wagtail admin panel(s)
FooModel.content_panels = [
    FieldPanel('title', classname="full title"),
    FieldPanel('introduction', classname="full"),
    FieldPanel('body', classname="full"),
    ]
```

```
# translation.py
from .models import Foo
from wagtail_modeltranslation.translation import TranslationOptions
from wagtail_modeltranslation.decorators import register

@register(Foo)
class FooTR(TranslationOptions):
 fields = (
    'title',
    'introduction',
    'body',
 )
```

2.5.1 Rules for Translated Field Access

The following rules apply to setting and getting the value of the original and the translation fields:

Rule 1

Reading the value from the original field returns the value translated to the current language.

Rule 2

Assigning a value to the original field updates the value in the associated current language translation field.

Rule 3

If both fields - the original and the current language translation field - are updated at the same time, the current language translation field wins.

Note: This can only happen in the model's constructor or `objects.create`. There is no other situation which can be considered *changing several fields at the same time*.

2.5.2 Examples of translated field access

Because the whole point of using the wagtail-modeltranslation app is translating dynamic content, the fields marked for translation are somehow special when it comes to accessing them. The value returned by a translated field is depending on the current language setting. **Language setting** refers to the Django `set_language` view and the corresponding `get_lang` function.

Assuming the current language is `pt` in the Foo example above, the translated `introduction` field will return the value from the `introduction_pt` field:

```
# Assuming the current language is "pt"
n = News.objects.all()[0]
t = n.introduction  # returns the Portuguese translation

# Assuming the current language is "pt"
t = n.introduction  # returns the Portuguese translation
```

This feature is implemented using Python descriptors making it happen without the need to touch the original model classes in any way. The descriptor uses the `django.utils.i18n.get_language` function to determine the current language.

2.6 Multilingual Manager

Every model registered for translation is patched so that all its managers become subclasses of `MultilingualManager` (of course, if a custom manager is defined on the model, its functions is retained). `MultilingualManager` simplifies language-aware queries, especially on third-party apps, by rewriting query field names.

Every model manager is patched, not only `objects` but also managers inherited from abstract base classes.

For example:

```
# Assuming the current language is "pt",
# these queries returns the same objects
Foo1 = Foo.objects.filter(introduction__contains='Welcome')
Foo2 = Foo.objects.filter(introduction_pt__contains='Welcome')

assert Foo1 == Foo2
```

It works this way: the translation field name is used, it is changed into the current language field name, based on the current language. Any language-suffixed fields are left untouched, so `title_es` wouldn't change, no matter what the current language is.

Rewriting of field names works with operators (like `__in`, `__ge`) as well as with relationship spanning. Moreover, it is also handled on `Q` and `F` expressions.

These manager methods perform rewriting:

- `filter()`, `exclude()`, `get()`
- `order_by()`
- `update()`
- `only()`, `defer()`
- `values()`, `values_list()`, with *fallback* mechanism
- `dates()`
- `select_related()`
- `create()`, with optional *auto-population* feature

In order not to introduce differences between `X.objects.create(...)` and `X(...)`, model constructor is also patched and performs rewriting of field names prior to regular initialization.

If one wants to turn rewriting of field names off, this can be easily achieved with `rewrite(mode)` method. `mode` is a boolean specifying whether rewriting should be applied. It can be changed several times inside a query. So `X.objects.rewrite(False)` turns rewriting off.

`MultilingualManager` offers one additional method: `raw_values`. It returns actual values from the database, without field names rewriting. Useful for checking translated field database value.

2.6.1 Auto-population

There is special manager method `populate mode `` which can trigger ``create()` or `get_or_create()` to populate all translation language fields with values from translated original ones. It can be very convenient when working with many languages. So:

```
x = Foo.objects.populate(True).create(title='bar')
```

is equivalent of:

```
x = Foo.objects.create(title_pt='bar', title_es='bar', title_fr='bar') ## title_?? for every language
```

Moreover, some fields can be explicitly assigned different values:

```
x = Foo.objects.populate(True).create(title='-- no translation yet --', title_es='hay traducción toda
```

It will result in `title_es == 'hay traducción todavía'` and other `title_?? == '-- no translation yet --'`.

There is another way of altering the current population status, an `auto_populate` context manager:

```
from modeltranslation.utils import auto_populate

with auto_populate(True):
    x = Foo.objects.create(title='bar')
```

Auto-population takes place also in model constructor, what is extremely useful when loading non-translated fixtures. Just remember to use the context manager:

```
with auto_populate():  # True can be ommited
   call_command('loaddata', 'fixture.json')  # Some fixture loading

   z = Foo(title='bar')
   print z.title_pt, z.title_es, z.title_fr  # prints 'bar bar bar'
```

There is a more convenient way than calling `populate` manager method or entering `auto_populate` manager context all the time: *Auto populate* setting. It controls the default population behaviour.

2.6.2 Auto-population modes

There are four different population modes:

`False` [set by default]

Auto-population turned off

`True` or `'all'` [default argument to population altering methods]

Auto-population turned on, copying translated field value to all other languages (unless a translation field value is provided)

'default' Auto-population turned on, copying translated field value to default language field (unless its value is provided)

'required' Acts like 'default', but copy value only if the original field is non-nullable

2.6.3 Falling back

Modeltranslation provides a mechanism to control behaviour of data access in case of empty translation values. This mechanism affects field access, as well as `values()` and `values_list()` manager methods.

Here is an example: a writer of some news hasn't specified a French title and content, but only the Spanish and Portuguese ones. Then if a French visitor is viewing the site, we would rather show him English title/content of the news than having empty strings displayed. This is called *fallback*.

```
news.title_en = 'English title'
news.title_fr = ''
print news.title
# If current active language is French, it should display the title_de field value (''),
# But if fallback is enabled, it would display 'English title' instead.

# Similarly for manager
news.save()
print News.objects.filter(pk=news.pk).values_list('title', flat=True)[0]
# As above: if current active language is French and fallback to English is enabled,
# it would display 'English title'.
```

There are several ways of controlling fallback, described below.

2.6.4 Fallback languages

Fallback languages setting allows to set the order of *fallback languages*. By default that's the `DEFAULT_LANGUAGE`.

For example, setting

```
MODELTRANSLATION_FALLBACK_LANGUAGES = ('en', 'es')
```

means: if current active language field value is unset, try English value. If it is also unset, try Portuguese, and so on - until some language yields a non-empty value of the field.

There is also an option to define a fallback by language, using dict syntax:

```
MODELTRANSLATION_FALLBACK_LANGUAGES = {
    'default': ('pt', 'es', 'en'),
    'fr': ('es',),
    'uk': ('fr',)
}
```

The `default` key is required and its value denote languages which are always tried at the end. With such a setting:

- for *uk* the order of fallback languages is: `('ru', 'en', 'de', 'fr')`
- for *fr* the order of fallback languages is: `('de', 'en')` - Note, that *fr* obviously is not a fallback, since its active language and *de* would be tried before *en*
- for *en* and *de* the fallback order is `('de', 'fr')` and `('en', 'fr')`, respectively
- for any other language the order of fallback languages is just `('en', 'de', 'fr')`

What is more, fallback languages order can be overridden per model, using `TranslationOptions`:

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    fallback_languages = {'default': ('fa', 'km')}  # use Persian and Khmer as fallback for News
```

Dict syntax is only allowed there.

Even more, all fallbacks may be switched on or off for just some exceptional block of code using:

```
from modeltranslation.utils import fallbacks

with fallbacks(False):
    # Work with values for the active language only
```

2.6.5 Fallback values

But what if current language and all fallback languages yield no field value? Then modeltranslation will use the field's *fallback value*, if one was defined.

Fallback values are defined in `TranslationOptions`, for example:

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    fallback_values = _('-- sorry, no translation provided --')
```

In this case, if title is missing in active language and any of fallback languages, news title will be `'-- sorry, no translation provided --'` (maybe translated, since gettext is used). Empty text will be handled in same way.

Fallback values can be also customized per model field:

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    fallback_values = {
        'title': _('-- sorry, this news was not translated --'),
        'text': _('-- please contact our translator (translator@example.com) --')
    }
```

If current language and all fallback languages yield no field value, and no fallback values are defined, then modeltranslation will use the field's default value.

2.6.6 Fallback undefined

Another question is what do we consider "no value", on what value should we fall back to other translations? For text fields the empty string can usually be considered as the undefined value, but other fields may have different concepts of empty or missing values.

Modeltranslation defaults to using the field's default value as the undefined value (the empty string for non-nullable `CharFields`). This requires calling `get_default` for every field access, which in some cases may be expensive.

If you'd like to fall back on a different value or your default is expensive to calculate, provide a custom undefined value (for a field or model):

```
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    fallback_undefined = {
        'title': 'no title',
        'text': None
    }
```

2.6.7 The State of the original field

As defined by the *Rules for Translated Field Access*, accessing the original field is guaranteed to work on the associated translation field of the current language. This applies to both, read and write operations.

The actual field value (which *can* still be accessed through instance.__dict__['original_field_name']‘‘) however has to be considered **undetermined** once the field has been registered for translation. Attempts to keep the value in sync with either the default or current language's field value has raised a boatload of unpredictable side effects in older versions of modeltranslation.

> **Warning:** Do not rely on the underlying value of the *original field* in any way!

InfoPortugal, S.A. - https://github.com/infoportugal

2.7 Authors

2.7.1 Core Committers

- Diogo Marques
- Rui Martins
- Eduardo Nogueira

2.7.2 Contributors

- Django-modeltranslation
- Django-linguo

2.8 Change Log

v0.3.5: - Fixed URL property return None

v0.3.4: - Fixed update_translation_fields: added support to foreign keys (#42)

v0.3.3: - Added ImageChooserPanel compability (#34) - use build_localized_fieldname insetead using "%s_%s" only (#40)

v0.3.2: - Fixed route() method issue causing invalid field lookup;

v0.3.1: - Add support to snippets, using SnippetsTranslationMixin

v0.3: - Fix conflicts in migrations with wagtailcore migrations. Now translated fields lives only on Page child classes tables;

v0.2.4: - Fix missing Site class import; - Fix missing reverse function import;

v0.2.3: - Add workaround for InlinePanel AttributeError (#31); - Added support to widget declarations on FieldPanel; - Fixed missing templatetags folder on pypi package;

v0.2.2: - Added duplicate content buttons to translated StreamFieldPanels;

v0.2.1: - Fixed missing templatetags folder on pypi package;

v0.2: - Support for StreamFields; - No more django-modeltranslation dependency;

v0.1.5

- Fixed required fields related bug

v0.1.4

- Support for wagtailsearch and wagtailsnippets

v0.1.3

- Support for translated inline panels #8: https://github.com/infoportugal/wagtail-modeltranslation/issues/8

v0.1.2

- Fixed Problem updating field with null value #6: https://github.com/infoportugal/wagtail-modeltranslation/issues/6

v0.1.1

- Fixed url_path issue caused by a browser with language different from settings.LANGUAGE_CODE

v0.1

- Minor release working but lacks full test coverage.
- Last version had required fields validation problems, now fixed.

v0.0.9

- Fixed issue that causes duplicated translation fields, preventing auto-slug from working properly.

v0.0.8

- Fixed issue related to deleting a non existing key on PAGE_EDIT_HANDLER dict

v0.0.7

- Fixed set_url_path() translatable model method

v0.0.6

- Fixed js issue related to auto-generating slugs

v0.0.5

- Now using django-modeltranslation 0.9.1;
- Fixed problem related to slug field fallbacks;

v0.0.4

** IMPORTANT: ** make sure that TranslationMixin comes before Page class on model inheritance

- Fix enhancement #1: url_path translation field
- Now includes a template tag that returns current page url to corresponding translated url
- New management command to update url_path translation fields - **set_translation_url_paths**

v0.0.3

- New methods;
- Now supports required fields;
- Fixed issue related to browser locale;

v0.0.2

- Prepopulated fields now works for translated fields (title and slug)