django-modeltranslation Documentation

Release 0.4.1

Dirk Eschler

May 16, 2016

Contents

The modeltranslation application can be used to translate dynamic content of existing Django 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 be able to add translations to existing or new projects and is fully integrated into the Django admin backend.

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, may they use translations or not, and you never have to touch the original model class.

CHAPTER 1

Features

- Unlimited number of target languages
- Add translations without changing existing models
- Django admin support
- Supports inherited models

1.1 Project Home

https://github.com/deschler/django-modeltranslation

1.2 Documentation

https://readthedocs.org/projects/django-modeltranslation

1.3 Mailing List

http://groups.google.com/group/django-modeltranslation

CHAPTER 2

Table of Contents

2.1 Installation

2.1.1 Requirements

Modeltranslation	Python	Django
>=0.4	2.5 - 2-7	1.3 - 1.4
<=0.3	2.4 - 2.7	1.0 - 1.4

2.1.2 Using Pip

```
$ pip install django-modeltranslation
```

2.1.3 Using the Source

Get a source tarball from github or pypi, unpack, then install with:

```
$ python setup.py install
```

Note: As an alternative, if you don't want to mess with any packaging tool, unpack the tarball and copy/move the modeltranslation directory to a path listed in your `PYTHONPATH` environment variable.

2.2 Setup

To setup the application please follow these steps. Each step is described in detail in the following sections:

1. Add the `modeltranslation` app to the `INSTALLED_APPS` variable of your project's `settings.py`.
2. Configure your `LANGUAGES` in `settings.py`.
3. Create a `translation.py` in your app directory and register `TranslationOptions` for every model you want to translate.
4. Sync the database using `manage.py syncdb` (note that this only applies if the models registered in the `translations.py` did not have been synced to the database before. If they did - read further down what to do in that case.

2.2.1 Configure the Project's `settings.py`

Required Settings

The following variables have to be added to or edited in the project's `settings.py`:

`INSTALLED_APPS`

Make sure that the `modeltranslation` app is listed in your `INSTALLED_APPS` variable:

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

Note: Also make sure that the app can be found on a path contained in your `PYTHONPATH` environment variable.

`LANGUAGES`

The `LANGUAGES` variable must contain all languages used for translation. The first language is treated as the *default language*.

The modeltranslation application uses the list of languages to add localized fields to the models registered for translation. To use the languages `de` and `en` in your project, set the `LANGUAGES` variable like this (where `de` is the default language):

```
gettext = lambda s: s
LANGUAGES = (
    ('de', gettext('German')),
    ('en', gettext('English')),
)
```

Note: The `gettext` lambda function is not a feature of modeltranslation, but rather required for Django to be able to (statically) translate the verbose names of the languages using the standard `i18n` solution.

Advanced Settings

Modeltranslation also has some advanced settings to customize its behaviour:

`MODELTRANSLATION_DEFAULT_LANGUAGE`

New in version 0.3.

Default: `None`

To override the default language as described in *LANGUAGES*, 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 = 'en'
```

MODELTRANSLATION_TRANSLATION_FILES

New in version 0.4.

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`.

A setting `MODELTRANSLATION_TRANSLATION_FILES` is provided to limit or 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',
)
```

Note: Modeltranslation up to version 0.3 used a single project wide registration file which was defined through `MODELTRANSLATION_TRANSLATION_REGISTRY = '<PROJECT_MODULE>.translation'`. For backwards compatibiliy the module defined through this setting is automatically added to `MODELTRANSLATION_TRANSLATION_FILES`. A `DeprecationWarning` is issued in this case.

MODELTRANSLATION_CUSTOM_FIELDS

Default: `()` (empty tuple)

New in version 0.3.

`Modeltranslation` officially supports `CharField` and `TextField`.

New in version 0.4.

Support for `FileField` and `ImageField`.

In most cases subclasses of the supported fields will work fine, too. Other fields aren't supported and will throw an `ImproperlyConfigured` exception.

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

Example:

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

Warning: This just prevents modeltranslation from throwing an `ImproperlyConfigured` exception. Any non text-like 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.

MODELTRANSLATION_DEBUG

Default: `settings.DEBUG`

New in version 0.4.

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 Registering Models for Translation

The `modeltranslation` app can translate `CharField` and `TextField` based fields (as well as `FileField` and `ImageField` as of version 0.4) of any model class. For each model to translate a translation option class containing the fields to translate is registered with the modeltranslation app.

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

1. Create a `translation.py` in your app directory.
2. Create a translation option class for every model to translate.
3. Register the model and the translation option class at the `modeltranslation.translator.translator`

The 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 fields of a model to translate. The class must derive from `modeltranslation.ModelTranslation` and it must provide a `fields` attribute storing the list of fieldnames. The option class must be registered with the `modeltranslation.translator.translator` instance.

To illustrate this let's have a look at a simple example using a `News` model. The news in this example only contains a `title` and a `text` field. Instead of a news, this could be any Django model class:

```
class News(models.Model):
    title = models.CharField(max_length=255)
    text = models.TextField()
```

In order to tell the modeltranslation app to translate the `title` and `text` field, create a `translation.py` file in your news app directory and add the following:

```
from modeltranslation.translator import translator, TranslationOptions
from news.models import News

class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)

translator.register(News, NewsTranslationOptions)
```

Note that this does not require to change the `News` model in any way, it's only imported. The `NewsTranslationOptions` derives from `TranslationOptions` and provides the `fields` attribute. Finally the model and its translation options are registered at the `translator` object.

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

2.3.1 Changes Automatically Applied to the Model Class

After registering the `News` model for translation an SQL dump of the news app will look like this:

```
$ ./manage.py sqlall news
BEGIN;
CREATE TABLE `news_news` (
    `id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `title` varchar(255) NOT NULL,
    `title_de` varchar(255) NULL,
    `title_en` varchar(255) NULL,
    `text` longtext NULL,
    `text_de` longtext NULL,
    `text_en` longtext NULL,
)
;
ALTER TABLE `news_news` ADD CONSTRAINT page_id_refs_id_3edd1f0d FOREIGN KEY (`page_id`) REFERENCES `p
CREATE INDEX `news_news_page_id` ON `news_news` (`page_id`);
COMMIT;
```

Note the `title_de`, `title_en`, `text_de` and `text_en` fields which are not declared in the original News model class but rather 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.

If you are starting a fresh project and have considered your translation needs in the beginning then simply sync your database and you are ready to use the translated models.

In case you are translating an existing project and your models have already been synced to the database you will need to alter the tables in your database and add these additional translation fields. Note that all added fields are declared `null=True` not matter if the original field is required. In other words - all translations are optional. To populate the default translation fields added by the modeltranslation application you can use the `update_translation_fields` command below. See *The update_translation_fields Command* section for more infos on this.

2.4 Accessing Translated and Translation Fields

The modeltranslation app changes the behaviour of the translated fields. To explain this consider the news example from the *Registering Models for Translation* chapter again. The original `News` model looked like this:

```
class News(models.Model):
    title = models.CharField(max_length=255)
    text = models.TextField()
```

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

```
class News(models.Model):
    title = models.CharField(max_length=255)  # original/translated field
    title_de = models.CharField(null=True, blank=True, max_length=255)  # default translation field
    title_en = models.CharField(null=True, blank=True, max_length=255)  # translation field
    text = models.TextField()  # original/translated field
    text_de = models.TextField(null=True, blank=True)  # default translation field
    text_en = models.TextField(null=True, blank=True)  # translation field
```

The example above assumes that the default language is `de`, therefore the `title_de` and `text_de` fields are marked as the *default translation fields*. If the default language is `en`, the `title_en` and `text_en` fields would be the *default translation fields*.

2.4.1 Rules for Translated Field Access

So now when it comes to setting and getting the value of the original and the translation fields the following rules apply:

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 also updates the value in the associated default translation field.

Rule 3

Assigning a value to the default translation field also updates the original field - note that the value of the original field will not be updated until the model instance is saved.

Rule 4

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

2.4.2 Examples for Translated Field Access

Because the whole point of using the 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" is referring to the Django set_language view and the corresponding `get_lang` function.

Assuming the current language is `de` in the News example from above, the translated `title` field will return the value from the `title_de` field:

```
# Assuming the current language is "de"
n = News.objects.all()[0]
t = n.title # returns german translation

# Assuming the current language is "en"
t = n.title # returns english 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.

2.5 Django Admin Integration

In order to be able to edit the translations via the `django.contrib.admin` application you need to register a special admin class for the translated models. The admin class must derive from `modeltranslation.admin.TranslationAdmin` which does some funky patching on all your models registered for translation. Taken the *news example* the most simple case would look like:

```
from django.contrib import admin
from news.models import News
from modeltranslation.admin import TranslationAdmin

class NewsAdmin(TranslationAdmin):
    pass

admin.site.register(News, NewsAdmin)
```

2.5.1 Tweaks Applied to the Admin

formfield_for_dbfield

The `TranslationBaseModelAdmin` class, which `TranslationAdmin` and all inline related classes in modeltranslation derive from, implements a special method which is `def formfield_for_dbfield(self, db_field, **kwargs)`. This method does the following:

1. Copies the widget of the original field to each of it's translation fields.
2. Checks if the original field was required and if so makes the default translation field required instead.

get_form/get_fieldsets/_declared_fieldsets

In addition the `TranslationBaseModelAdmin` class overrides `get_form`, `get_fieldsets` and `_declared_fieldsets` to make the options `fields`, `exclude` and `fieldsets` work in a transparent way. It basically does:

1. Removes the original field from every admin form by adding it to `exclude` under the hood.
2. Replaces the - now removed - orginal fields with their corresponding translation fields.

Taken the `fieldsets` option as an example, where the `title` field is registered for translation but not the `news` field:

```
class NewsAdmin(TranslationAdmin):
    fieldsets = [
        (u'News', {'fields': ('title', 'news',)})
    ]
```

In this case `get_fieldsets` will return a patched fieldset which contains the translation fields of `title`, but not the original field:

```
>>> a = NewsAdmin(NewsModel, site)
>>> a.get_fieldsets(request)
[(u'News', {'fields': ('title_de', 'title_en', 'news',)})]
```

2.5.2 TranslationAdmin in Combination with Other Admin Classes

If there already exists a custom admin class for a translated model and you don't want or can't edit that class directly there is another solution.

Taken a (fictional) reusable blog app which defines a model `Entry` and a corresponding admin class called `EntryAdmin`. This app is not yours and you don't want to touch it at all.

In the most common case you simply make use of Python's support for multiple inheritance like this:

```
class MyTranslatedEntryAdmin(EntryAdmin, TranslationAdmin):
    pass
```

The class is then registered for the `admin.site` (not to be confused with modeltranslation's `translator`). If `EntryAdmin` is already registered through the blog app, it has to be unregistered first:

```
admin.site.unregister(Entry)
admin.site.register(Entry, MyTranslatedEntryAdmin)
```

Admin Classes that Override `formfield_for_dbfield`

In a more complex setup the original `EntryAdmin` might override `formfield_for_dbfield` itself:

```
class EntryAdmin(model.Admin):
    def formfield_for_dbfield(self, db_field, **kwargs):
        # does some funky stuff with the formfield here
```

Unfortunately the first example won't work anymore because Python can only execute one of the `formfield_for_dbfield` methods. Since both admin classes implement this method Python must make a decision and it chooses the first class `EntryAdmin`. The functionality from `TranslationAdmin` will not be executed and translation in the admin will not work for this class.

But don't panic, here's a solution:

```
class MyTranslatedEntryAdmin(EntryAdmin, TranslationAdmin):
    def formfield_for_dbfield(self, db_field, **kwargs):
        field = super(MyTranslatedEntryAdmin, self).formfield_for_dbfield(db_field, **kwargs)
        self.patch_translation_field(db_field, field, **kwargs)
        return field
```

This implements the `formfield_for_dbfield` such that both functionalities will be executed. The first line calls the superclass method which in this case will be the one of `EntryAdmin` because it is the first class inherited from. The `TranslationAdmin` capsulates its functionality in the `patch_translation_field` method and the `formfield_for_dbfield` implementation of the `TranslationAdmin` class simply calls it. You can copy this behaviour by calling it from a custom admin class and that's done in the example above. After that the `field` is fully patched for translation and finally returned.

2.5.3 Admin Inlines

New in version 0.2.

Support for tabular and stacked inlines, common and generic ones.

A translated inline must derive from one of the following classes:

- `modeltranslation.admin.TranslationTabularInline`
- `modeltranslation.admin.TranslationStackedInline`

- modeltranslation.admin.TranslationGenericTabularInline
- modeltranslation.admin.TranslationGenericStackedInline

Just like TranslationAdmin these classes implement a special method formfield_for_dbfield which does all the patching.

For our example we assume that there is new model called Image. Its definition is left out for simplicity. Our News model inlines the new model:

```
from django.contrib import admin
from news.models import Image, News
from modeltranslation.admin import TranslationTabularInline

class ImageInline(TranslationTabularInline):
    model = Image

class NewsAdmin(admin.ModelAdmin):
    list_display = ('title',)
    inlines = [ImageInline,]

admin.site.register(News, NewsAdmin)
```

Note: In this example only the Image model is registered in translation.py. It's not a requirement that NewsAdmin derives from TranslationAdmin in order to inline a model which is registered for translation.

Complex Example with Admin Inlines

In this more complex example we assume that the News and Image models are registered in translation.py. The News model has an own custom admin class called NewsAdmin and the Image model an own generic stacked inline class called ImageInline. Furthermore we assume that NewsAdmin overrides formfield_for_dbfield itself and the admin class is already registered through the news app.

Note: The example uses the technique described in *TranslationAdmin in combination with other admin classes*.

Bringing it all together our code might look like this:

```
from django.contrib import admin
from news.admin import ImageInline
from news.models import Image, News
from modeltranslation.admin import TranslationAdmin, TranslationGenericStackedInline

class TranslatedImageInline(ImageInline, TranslationGenericStackedInline):
    model = Image

class TranslatedNewsAdmin(NewsAdmin, TranslationAdmin):
    inlines = [TranslatedImageInline,]

    def formfield_for_dbfield(self, db_field, **kwargs):
        field = super(TranslatedNewsAdmin, self).formfield_for_dbfield(db_field, **kwargs)
        self.patch_translation_field(db_field, field, **kwargs)
        return field

admin.site.unregister(News)
admin.site.register(News, NewsAdmin)
```

2.5.4 Using Tabbed Translation Fields

New in version 0.3.

Modeltranslation supports separation of translation fields via jquery-ui tabs. The proposed way to include it is through the inner `Media` class of a `TranslationAdmin` class like this:

```
class NewsAdmin(TranslationAdmin):
    class Media:
        js = (
            'modeltranslation/js/force_jquery.js',
            'http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.24/jquery-ui.min.js',
            'modeltranslation/js/tabbed_translation_fields.js',
        )
        css = {
            'screen': ('modeltranslation/css/tabbed_translation_fields.css',),
        }
```

The `force_jquery.js` script is necessary when using Django's built-in `django.jQuery` object. Otherwise the *normal* `jQuery` object won't be available to the included (non-namespaced) jquery-ui library.

Standard jquery-ui theming can be used to customize the look of tabs, the provided css file is supposed to work well with a default Django admin.

Note: This is just an example and might have to be adopted to your setup.

2.5.5 Using a Custom jQuery Library

If you don't want to use the jquery library shipped with Django, you can also include a standard one. While this adds some redundancy it could be useful in situations where you need certain features from a newer version of jquery that is not yet included in Django or you rely on a non-namespaced version of jquery somewhere in your custom admin frontend code or included plugins.

In this case you don't need the `force_jquery.js` static provided by modeltranslation but include the standard jquery library before jquery-ui like this:

```
class NewsAdmin(TranslationAdmin):
    class Media:
        js = (
            'http://code.jquery.com/jquery-1.8.2.min.js',
            'http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.24/jquery-ui.min.js',
            'modeltranslation/js/tabbed_translation_fields.js',
        )
        css = {
            'screen': ('modeltranslation/css/tabbed_translation_fields.css',),
        }
```

2.6 Management Commands

2.6.1 The `update_translation_fields` Command

In case the modeltranslation app was installed on an existing project and you have specified to translate fields of models which are already synced to the database, you have to update your database schema manually.

Unfortunately the newly added translation fields on the model will be empty then, and your templates will show the translated value of the fields (see Rule 1 below) which will be empty in this case. To correctly initialize the default translation field you can use the `update_translation_fields` command:

```
$ ./manage.py update_translation_fields
```

Taken the News example from above this command will copy the value from the news object's `title` field to the default translation field `title_de`. It only does so if the default translation field is empty otherwise nothing is copied.

Note: The command will examine your `settings.LANGUAGES` variable and the first language declared there will be used as the default language.

All translated models (as specified in the project's `translation.py` will be populated with initial data.

2.6.2 The `sync_translation_fields` Command

New in version 0.4.

```
$ ./manage.py sync_translation_fields
```

Todo

Explain

2.7 Caveats

Consider the following example (assuming the default language is `de`):

```
>>> n = News.objects.create(title="foo")
>>> n.title
'foo'
>>> n.title_de
>>>
```

Because the original field `title` was specified in the constructor it is directly passed into the instance's `__dict__` and the descriptor which normally updates the associated default translation field (`title_de`) is not called. Therefor the call to `n.title_de` returns an empty value.

Now assign the title, which triggers the descriptor and the default translation field is updated:

```
>>> n.title = 'foo'
>>> n.title_de
'foo'
>>>
```

2.7.1 Accessing Translated Fields Outside Views

Since the modeltranslation mechanism relies on the current language as it is returned by the `get_language` function care must be taken when accessing translated fields outside a view function.

Within a view function the language is set by Django based on a flexible model described at How Django discovers language preference which is normally used only by Django's static translation system.

When a translated field is accessed in a view function or in a template, it uses the `django.utils.translation.get_language` function to determine the current language and return the appropriate value.

Outside a view (or a template), i.e. in normal Python code, a call to the `get_language` function still returns a value, but it might not what you expect. Since no request is involved, Django's machinery for discovering the user's preferred language is not activated.

Todo

Explain more

The unittests in `tests.py` use the `django.utils.translation.trans_real` functions to activate and deactive a specific language outside a view function.

2.8 Related Projects

Note: This list is horribly outdated and only covers apps that where available when modeltranslation was initially developed. A more complete list can be found at djangopackages.com.

2.8.1 django-multilingual

A library providing support for multilingual content in Django models.

It is not possible to reuse existing models without modifying them.

2.8.2 django-multilingual-model

A much simpler version of the above *django-multilingual*.

It works very similiar to the *django-multilingual* approach.

2.8.3 transdb

Django's field that stores labels in more than one language in database.

This approach uses a specialized `Field` class, which means one has to change existing models.

2.8.4 i18ndynamic

This approach is not developed any more.

2.8.5 django-pluggable-model-i18n

This app utilizes a new approach to multilingual models based on the same concept the new admin interface uses. A translation for an existing model can be added by registering a translation class for that model.

This is more or less what modeltranslation does, unfortunately it is far from being finished.

2.9 Authors

- Peter Eschler <peschler@gmail.com>
- Dirk Eschler <eschler@gmail.com>

2.10 Contributors

- Carl J. Meyer
- Jaap Roes
- Bojan Mihelac
- Sébastien Fievet
- Jacek Tomaszewski
- Bruno Tavares
- And many more ...