Contents

1 Features
  1.1 Project Home ................................................................. 3
  1.2 Documentation ................................................................ 3
  1.3 Mailing List .................................................................. 3

2 Table of Contents
  2.1 Installation .................................................................... 5
  2.2 Setup ........................................................................... 6
  2.3 Configuration .................................................................. 6
  2.4 Registering Models for Translation ......................................... 11
  2.5 Accessing Translated and Translation Fields .............................. 16
  2.6 ModelForms .................................................................. 22
  2.7 Django Admin Integration ................................................... 24
  2.8 Management Commands ..................................................... 28
  2.9 Caveats ......................................................................... 30
  2.10 How to Contribute ............................................................ 31
  2.11 Related Projects .............................................................. 32
  2.12 ChangeLog .................................................................. 33

3 Authors
  3.1 Core Committers .............................................................. 35
  3.2 Contributors ................................................................ 35

Index ............................................................................... 37
The modeltranslation application is 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

• 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
• Django admin integration
• Flexible fallbacks, auto-population and more!

1.1 Project Home

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

1.2 Documentation

https://django-modeltranslation.readthedocs.org/en/latest

1.3 Mailing List

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

2.1.1 Requirements

Which Modeltranslation version is required for given Django-Python combination to work?

<table>
<thead>
<tr>
<th>Python version</th>
<th>1.8</th>
<th>1.9</th>
<th>1.10</th>
<th>1.11</th>
<th>2.0</th>
<th>2.1</th>
<th>2.2</th>
<th>3.0</th>
<th>3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>0.9+</td>
<td></td>
<td>0.12+</td>
<td>0.13+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>0.9+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>0.9+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>0.9+</td>
<td>0.11+</td>
<td>0.12+</td>
<td>0.13+</td>
<td>0.13+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>0.9+</td>
<td>0.11+</td>
<td>0.12+</td>
<td>0.13+</td>
<td>0.13+</td>
<td>0.13+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.13+</td>
<td>0.13+</td>
<td>0.15+</td>
<td>0.15+</td>
<td>0.17+</td>
</tr>
<tr>
<td>3.7</td>
<td></td>
<td></td>
<td></td>
<td>0.13+</td>
<td>0.13+</td>
<td>0.15+</td>
<td>0.15+</td>
<td>0.17+</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td></td>
<td></td>
<td></td>
<td>0.13+</td>
<td>0.13+</td>
<td>0.15+</td>
<td>0.15+</td>
<td>0.17+</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td></td>
<td></td>
<td></td>
<td>0.13+</td>
<td>0.13+</td>
<td>0.15+</td>
<td>0.15+</td>
<td>0.17+</td>
<td></td>
</tr>
</tbody>
</table>

(−X denotes “up to version X”, whereas X+ means “from version X upwards”)

2.1.2 Using Pip

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

2.1.3 Using the Source

Get a source tarball from pypi, unpack, then install with:
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 `modeltranslation` to the `INSTALLED_APPS` variable of your project’s `settings.py`.
2. Set `USE_I18N = True` in `settings.py`.
3. Configure your `LANGUAGES` in `settings.py`.
4. Create a `translation.py` in your app directory and register `TranslationOptions` for every model you want to translate.
5. Sync the database using `python manage.py makemigrations` and `python manage.py migrate`.

Note: This only applies if the models registered in `translation.py` haven’t been synced to the database before. If they have, please read Committing fields to database.

Note: If you are using Django 1.7 and its internal migration system, run `python manage.py makemigrations`, followed by `python manage.py migrate` instead. See Migrations (Django 1.7) for details.

2.3 Configuration

2.3.1 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:

```python
INSTALLED_APPS = (  
    ...
    'modeltranslation',
    'django.contrib.admin', # optional
    ...
)
```
Important: If you want to use the admin integration, `modeltranslation` must be put before `django.contrib.admin` (only applies when using Django 1.7 or above).

Important: If you want to use the `django-debug-toolbar` together with `modeltranslation`, use explicit setup. Otherwise tweak the order of `INSTALLED_APPS`: try to put `debug_toolbar` as first entry in `INSTALLED_APPS` (in Django < 1.7) or after `modeltranslation` (in Django >= 1.7). However, only explicit setup is guaranteed to succeed.

LANGUAGES

The `LANGUAGES` variable must contain all languages used 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. To use the languages `de` and `en` in your project, set the `LANGUAGES` variable like this (where `de` is the default language):

```python
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.

Note: If, for some reason, you don’t want to translate objects to exactly the same languages as the site would be displayed into, you can set `MODELTRANSLATION_LANGUAGES` (see below). For any language in `LANGUAGES` not present in `MODELTRANSLATION_LANGUAGES`, the default language will be used when accessing translated content. For any language in `MODELTRANSLATION_LANGUAGES` not present in `LANGUAGES`, probably nobody will see translated content, since the site wouldn’t be accessible in that language.

Warning: Modeltranslation does not enforce the `LANGUAGES` setting to be defined in your project. When it isn’t present (and neither is `MODELTRANSLATION_LANGUAGES`), it defaults to Django’s global `LANGUAGES` setting instead, and that are quite a few languages!

2.3.2 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_LANGUAGES**

New in version 0.8.

Default: same as `LANGUAGES`

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

Example:

```
LANGUAGES = (
    ('en', 'English'),
    ('de', 'German'),
    ('pl', 'Polish'),
)
MODELTRANSLATION_LANGUAGES = ('en', 'de')
```

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

**MODELTRANSLATION_FALLBACK_LANGUAGES**

New in version 0.5.

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 = ('en', 'de')
```

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': ('en', 'de'), 'fr': ('de',)}
```
**Note:** Each language has to be in the `LANGUAGES` setting, otherwise an `ImproperlyConfigured` exception is raised.

### `MODELTRANSLATION_PREPOPULATE_LANGUAGE`

New in version 0.7.

Default: 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:

```python
MODELTRANSLATION_PREPOPULATE_LANGUAGE = 'en'
```

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

### `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`.

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

**Syntax:**

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

Example:

```python
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'`. In version 0.4 and 0.5, for backwards compatibility, the module defined through this setting was automatically added to `MODELTRANSLATION_TRANSLATION_FILES`. A `DeprecationWarning` was issued in this case.
In version 0.6, \texttt{MODELTRANSLATION\_TRANSLATION\_REGISTRY} is handled no more.

\textbf{MODELTRANSLATION\_CUSTOM\_FIELDS}

Default: (empty tuple)
New in version 0.3.
Modeltranslation supports the fields listed in the \textit{Supported Fields Matrix}. In most cases subclasses of the supported fields will work fine, too. Unsupported fields will throw an \texttt{ImproperlyConfigured} exception.
The list of supported fields can be extended by defining a tuple of field names in your \texttt{settings.py}.
Example:
\begin{verbatim}
MODELTRANSLATION\_CUSTOM\_FIELDS = ('MyField', 'MyOtherField',)
\end{verbatim}

\textbf{Warning:} This just prevents modeltranslation from throwing an \texttt{ImproperlyConfigured} 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.

\textbf{MODELTRANSLATION\_AUTO\_POPULATE}

Default: False
New in version 0.5.
This setting controls if the \textit{Multilingual Manager} should automatically populate language field values in its \texttt{create} and \texttt{get\_or\_create} method, and in model constructors, so that these two blocks of statements can be considered equivalent:

\begin{verbatim}
News\_objects\_populate(True).create(title='-- no translation yet --')
with auto\_populate(True):
    q = News(title='-- no translation yet --')
# same effect with MODELTRANSLATION\_AUTO\_POPULATE == True:
News\_objects\_create(title='-- no translation yet --')
q = News(title='-- no translation yet --')
\end{verbatim}

Possible modes are listed \textit{here}.

\textbf{MODELTRANSLATION\_DEBUG}

Default: False
New in version 0.4.
Changed in version 0.7.
Used for modeltranslation related debug output. Currently setting it to \texttt{False} will just prevent Django’s development server from printing the \texttt{Registered xx models for translation message} to stdout.
MODELTRANSLATION_ENABLE_FALLBACKS

Default: True
New in version 0.6.
Control if fallback (both language and value) will occur.

MODELTRANSLATION_LOADDATA_RETAIN_LOCALE

Default: True
New in version 0.7.
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. For each model to translate, a translation option class containing the fields to translate is registered with a special object called the translator.

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 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.translator.TranslationOptions 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:

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

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

```python
from modeltranslation.translator import translator, TranslationOptions
from .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.

New in version 0.10.

If you prefer, register is also available as a decorator, much like the one Django introduced for its admin in version 1.7. Usage is similar to the standard register, just provide arguments as you normally would, except the options class which will be the decorated one:

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

@register(News)
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
```

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

### 2.4.1 TranslationOptions fields inheritance

New in version 0.5.

A subclass of any TranslationOptions will inherit fields from its bases (similar to the way Django models inherit fields, but with a different syntax).

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

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

class NewsWithImageTranslationOptions(NewsTranslationOptions):
    fields = ('image',)

translator.register(News, NewsTranslationOptions)
translator.register(NewsWithImage, NewsWithImageTranslationOptions)
```

The above example adds the fields title and text from the NewsTranslationOptions class to NewsWithImageTranslationOptions, or to say it in code:

```python
assert NewsWithImageTranslationOptions.fields == ('title', 'text', 'image')
```

Of course multiple inheritance and inheritance chains (A > B > C) also work as expected.

**Note:** When upgrading from a previous modeltranslation version (<0.5), please review your TranslationOptions classes and see if introducing fields inheritance broke the project (if you had always subclassed TranslationOptions only, there is no risk).

### 2.4.2 Changes Automatically Applied to the Model Class

After registering the News model for translation a SQL dump of the news app will look like this:
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.py.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.

### 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` gather 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.

### Committing fields to database

If you are starting a fresh project and have considered your translation needs in the beginning then simply sync your database (``./manage.py syncdb`` or ``./manage.py schemamigration myapp --initial`` if using South) 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. If you are using South, you’re done: simply create a new migration (South will detect newly added translation fields) and apply it. If not, you can use a little helper: `The sync_translation_fields Command` which can execute schema-ALTERing SQL to add new fields. Use either of these two solutions, not both.

If you are adding translation fields to a third-party app that is using South, things get more complicated. In order to be able to update the app in the future, and to feel comfortable, you should use the `sync_translation_fields` command.
command. Although it’s possible to introduce new fields in a migration, it’s nasty and involves copying migration files, using SOUTH_MIGRATION_MODULES setting, and passing --delete-ghost-migrations flag, so we don’t recommend it. Invoking sync_translation_fields is plain easier.

Note that all added fields are by default declared blank=True and null=True no matter if the original field is required or not. In other words - all translations are optional, unless an explicit option is provided - see Required fields.

To populate the default translation fields added by modeltranslation with values from existing database fields, you can use the update_translation_fields command. See The update_translation_fields Command for more info on this.

Migrations (Django 1.7)

New in version 0.8.

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.MODELTRANSLATIONLANGUAGES.
- Registered or unregistered a field through TranslationOptions.fields.

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

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

Note: Support for migrations is implemented through fields.TranslationField.deconstruct(self) and respects changes to the null option.

2.4.3 Required fields

New in version 0.8.

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

```python
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    required_languages = ('en', 'de')
```

It’s quite self-explanatory: for German and English, all translation fields are required. For other languages - optional.

A more fine-grained control is available:

```python
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    required_languages = {'de': ('title', 'text'), 'default': ('title',)}
```

For German, all fields (both title and text) are required; for all other languages - only title is required. The 'default' is optional.
Note: Requirement is enforced by `blank=False`. Please remember that it will trigger validation only in model forms 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.

### 2.4.4 TranslationOptions attributes reference

Quick cheatsheet with links to proper docs sections and examples showing expected syntax.

Classes inheriting from `TranslationOptions` can have following attributes defined:

- **TranslationOptions.fields** *(required)*
  - List of translatable model fields. See [Registering Models for Translation](#).

```python
Some fields can be implicitly added through inheritance, see TranslationOptions fields inheritance.
```

- **TranslationOptions.fallback_languages**
  - Control order of languages for fallback purposes. See [Fallback languages](#).

```python
fallback_languages = {'default': ('en', 'de', 'fr'), 'uk': ('ru',)}
```

- **TranslationOptions.fallback_values**
  - Set the value that should be used if no fallback language yielded a value. See [Fallback values](#).

```python
fallback_values = _('-- sorry, no translation provided --')
fallback_values = {'title': _('Object not translated'), 'text': '---'}
```

- **TranslationOptions.fallback_undefined**
  - Set what value should be considered "no value". See [Fallback undefined](#).

```python
fallback_undefined = None
```

- **TranslationOptions.empty_values**
  - Override the value that should be saved in forms on empty fields. See [Formfields and nullability](#).

```python
empty_values = ''
empty_values = {'title': '', 'slug': None, 'desc': 'both'}
```

- **TranslationOptions.required_languages**
  - Control which translation fields are required. See [Required fields](#).

```python
required_languages = ('en', 'de')
required_languages = ('de': {'title', 'text'}, 'default': {'title',})
```

### 2.4.5 Supported Fields Matrix

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 gives an overview about their current support status:
<table>
<thead>
<tr>
<th>Model Field</th>
<th>0.4</th>
<th>0.5</th>
<th>0.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoField</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BigIntegerField</td>
<td>No</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>BooleanField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CharField</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CommaSeparatedIntegerField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DateField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DateTimeField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DecimalField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EmailField</td>
<td>Yes*</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>FileField</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FilePathField</td>
<td>Yes*</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>FloatField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ImageField</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IntegerField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IPAddressField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GenericIPAddressField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NullBooleanField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PositiveIntegerField</td>
<td>No</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>PositiveSmallIntegerField</td>
<td>No</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>SlugField</td>
<td>Yes*</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>SmallIntegerField</td>
<td>No</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>TextField</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TimeField</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>URLField</td>
<td>Yes*</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>ForeignKey</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>OneToOneField</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ManyToManyField</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* Implicitly supported (as subclass of a supported field)

### 2.5 Accessing Translated and Translation Fields

Modeltranslation 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:

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

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

```python
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
```

(continues on next page)
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.

2.5.1 Rules for Translated Field Access

Changed in version 0.5.

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 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 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:

```python
# 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.

Todo: Add more examples.
2.5.3 Multilingual Manager

New in version 0.5.

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

Every model’s manager is patched, not only objects (even managers inherited from abstract base classes).

For example:

```python
# Assuming the current language is "de",
# these queries returns the same objects
news1 = News.objects.filter(title__contains='enigma')
news2 = News.objects.filter(title_de__contains='enigma')
assert news1 == news2
```

It works as follow: if the translation field name is used (title), it is changed into the current language field name (title_de or title_en, depending on the current active language). Any language-suffixed names are left untouched (so title_en 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.

Auto-population

Changed in version 0.6.

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 = News.objects.populate(True).create(title='bar')

is equivalent of:

x = News.objects.create(title_en='bar', title_de='bar') # title_?? for every language

Moreover, some fields can be explicitly assigned different values:

x = News.objects.populate(True).create(title='-- no translation yet --', title_de='enigma')

It will result in title_de == 'enigma' and other title_?? == '-- no translation yet --'.

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

```python
from modeltranslation.utils import auto_populate

with auto_populate(True):
    x = News.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:

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

    z = News(title='bar')
    print z.title_en, z.title_de  # prints 'bar bar'
```

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

### 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

#### 2.5.4 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.
Consider the News example: a creator of some news hasn’t specified its German title and content, but only English ones. Then if a German visitor is viewing the site, we would rather show him English title/content of the news than display empty strings. This is called fallback.

```python
news.title_en = 'English title'
news.title_de = ''
print news.title
# If current active language is German, 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 German and fallback to English is enabled, it would display 'English title'.
```

There are several ways of controlling fallback, described below.

## Fallback languages

New in version 0.5.

The `MODELTRANSLATION_FALLBACK_LANGUAGES` setting allows to set the order of fallback languages. By default that’s the `DEFAULT_LANGUAGE`.

For example, setting

```python
MODELTRANSLATION_FALLBACK_LANGUAGES = ('en', 'de', 'fr')
```

means: if current active language field value is unset, try English value. If it is also unset, try German, 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:

```python
MODELTRANSLATION_FALLBACK_LANGUAGES = {
    'default': ('en', 'de', 'fr'),
    'fr': ('de',),
    'uk': ('ru',)
}
```

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`:

```python
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.

New in version 0.6.

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

```python
from modeltranslation.utils import fallbacks

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

### Fallback values

New in version 0.4.

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:

```python
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:

```python
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.

### Fallback undefined

New in version 0.7.

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):

```python
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    fallback_undefined = {
        'title': 'no title',
    }
```

(continues on next page)
2.5.5 The State of the Original Field

Changed in version 0.5.

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.

Since version 0.12 the original field is expected to have even more undetermined value. It’s because Django 1.10 changed the way deferred fields work.

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

**Todo:** Perhaps outline effects this might have on the `update_translation_field` management command.

2.6 ModelForms

ModelForms for multilanguage models are defined and handled as typical ModelForms. Please note, however, that they shouldn’t be defined next to models (see a note).

Editing multilanguage models with all translation fields in the admin backend is quite sensible. However, presenting all model fields to the user on the frontend may be not the right way. Here comes the TranslationModelForm which strip out all translation fields:

```python
from news.models import News
from modeltranslation.forms import TranslationModelForm

class MyForm(TranslationModelForm):
    class Meta:
        model = News
```

Such a form will contain only original fields (title, text - see example). Of course, upon saving, provided values would be set on proper attributes, depending on the user current language.

2.6.1 Formfields and nullability

New in version 0.7.1.
Note: Please remember that all translation fields added to model definition are nullable (null=True), regardless of the original field nullability.

In most cases formfields for translation fields behave as expected. However, there is one annoying problem with `models.CharField` - probably the most commonly translated field type.

The problem is that default formfield for `CharField` stores empty values as empty strings (''), even if the field is nullable (see django ticket #9590).

Thus formfields for translation fields are patched by modeltranslation. The following rules apply:

- If the original field is not nullable, an empty value is saved as '';
- If the original field is nullable, an empty value is saved as `None`.

To deal with complex cases, these rules can be overridden per model or even per field using `TranslationOptions`:

```python
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    empty_values = None

class ProjectTranslationOptions(TranslationOptions):
    fields = ('name', 'slug', 'description',)
    empty_values = {'name': '', 'slug': None}
```

If a field is not mentioned while using dict syntax, the default rules apply.

This configuration is especially useful for fields with unique constraints:

```python
class Category(models.Model):
    name = models.CharField(max_length=40)
    slug = models.SlugField(max_length=30, unique=True)
```

Because the `slug` field is not nullable, its translation fields would store empty values as '' and that would result in an error when two or more `Categories` are saved with `slug_en` empty - unique constraints wouldn’t be satisfied. Instead, `None` should be stored, as several `None` values in the database don’t violate uniqueness:

```python
class CategoryTranslationOptions(TranslationOptions):
    fields = ('name', 'slug')
    empty_values = {'slug': None}
```

**None-checkbox widget**

Maybe there is a situation where you want to store both - empty strings and `None` values - in a field. For such a scenario there is a third configuration value: 'both':

```python
class NewsTranslationOptions(TranslationOptions):
    fields = ('title', 'text',)
    empty_values = {'title': None, 'text': 'both'}
```

It results in a special widget with a None-checkbox to null a field. It’s not recommended in frontend as users may be confused what this `None` is. The only useful place for this widget might be the admin backend; see Formfields with None-checkbox.

To sum it up, the valid values for `empty_values` are: `None`, '' and 'both'.

2.6. ModelForms
2.7 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:

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

class NewsAdmin(TranslationAdmin):
    pass

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

2.7.1 Tweaks Applied to the Admin

`formfield_for_dbfield`

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

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

`get_form/get_fieldsets`

In addition the `TranslationBaseModelAdmin` class overrides `get_form` and `get_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 - original 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:

```python
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:

```python
>>> a = NewsAdmin(NewsModel, site)
>>> a.get_fieldsets(request)
[(u'News', {'fields': ('title_de', 'title_en', 'news',)})]
```
2.7.2 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 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:

```python
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:

```python
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:

```python
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:

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

2.7.3 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
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 a new model called `Image`. The definition is left out for simplicity. Our `News` model inlines the new model:

```python
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.`
2.7.4 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:

```python
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',}
        }
```

Note: Here we stick to the jquery library shipped with Django. 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.

As an alternative, if want to use a more recent version of jquery, you can do so by including this in your Media class instead:

```python
class NewsAdmin(TranslationAdmin):
    class Media:
        js = [
            'http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js',
            'http://ajax.googleapis.com/ajax/libs/jqueryui/1.10.2/jquery-ui.min.js',
            'modeltranslation/js/tabbed_translation_fields.js',
        ]
        css = {
            'screen': {'modeltranslation/css/tabbed_translation_fields.css',}
        }
```

Tabbed Translation Fields Admin Classes

New in version 0.7.

To ease the inclusion of the required static files for tabbed translation fields, the following admin classes are provided:

- TabbedDjangoJqueryTranslationAdmin (aliased to TabbedTranslationAdmin)
- TabbedExternalJqueryTranslationAdmin
Rather than inheriting from `TranslationAdmin`, simply subclass one of these classes like this:

```python
class NewsAdmin(TabbedTranslationAdmin):
    pass
```

### 2.7.5 TranslationAdmin Options

#### TranslationAdmin.group_fieldsets

New in version 0.6.

When this option is activated untranslated and translation fields are grouped into separate fieldsets. The first fieldset contains the untranslated fields, followed by a fieldset for each translation field. The translation field fieldsets use the original field’s `verbose_name` as a label.

Activating the option is a simple way to reduce the visual clutter one might experience when mixing these different types of fields.

The `group_fieldsets` option expects a boolean. By default fields are not grouped into fieldsets (`group_fieldsets = False`).

A few simple policies are applied:

- A `fieldsets` option takes precedence over the `group_fieldsets` option.
- Other default `ModelAdmin` options like `exclude` are respected.

```python
class NewsAdmin(TranslationAdmin):
    group_fieldsets = True
```

#### Formfields with None-checkbox

There is the special widget which allow to choose whether empty field value should be stores as empty string or None (see `None-checkbox widget`). In `TranslationAdmin` some fields can use this widget regardless of their `empty_values` setting:

```
class NewsAdmin(TranslationAdmin):
    both_empty_values_fields = ('title', 'text')
```

### 2.8 Management Commands

#### 2.8.1 The `update_translation_fields` Command

In case modeltranslation was installed in 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 (see `Committing fields to database`).

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`) which will be empty in this case. To correctly initialize the default translation field you can use the `update_translation_fields` command:

```
$ python manage.py update_translation_fields
```
Taken the news example used throughout the documentation this command will copy the value from the news object’s `title` field to the translation field `title_de`. It only does so if the translation field is empty otherwise nothing is copied.

On default, only the default language will have its translation field populated, but you can provide a --language option to specify any other language listed in `settings.py`.

**Note:** Unless you configured modeltranslation to override the default language 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 translation files) from all apps will be populated with initial data. Optionally, an app label and model name may be passed to populate only a subset of translated models.

```bash
$ python manage.py update_translation_fields myapp

$ python manage.py update_translation_fields myapp mymodel
```

### 2.8.2 The `sync_translation_fields` Command

New in version 0.4.

```bash
$ python manage.py sync_translation_fields
```

This command compares the database and translated models definitions (finding new translation fields) and provides SQL statements to alter tables. You should run this command after adding a new language to your `settings.LANGUAGES` or a new field to the `TranslationOptions` of a registered model.

However, if you are using South in your project, in most cases it’s recommended to use migration instead of `sync_translation_fields`. See [Committing fields to database](#) for detailed info and use cases.

### 2.8.3 The `loaddata` Command

New in version 0.7.

An extended version of Django’s original `loaddata` command which adds an optional `populate` keyword. If the keyword is specified, the normal loading command will be run under the selected auto-population modes.

By default no auto-population is performed.

```bash
$ python manage.py loaddata --populate=all fixtures.json

$ python manage.py loaddata --populate=false fixtures.json
$ python manage.py loaddata --populate=0 fixtures.json
```

**Note:** If `populate` is not specified, the current auto-population mode is used. `Current` means the one set by `settings`.

Moreover, this `loaddata` command version can override the nasty habit of changing locale to `en-us`. By default, it will retain the proper locale. To get the old behaviour back, set `MODELTRANSLATION_LOADDATA_RETAIN_LOCALE` to `False`.

2.8. Management Commands 29
2.9 Caveats

2.9.1 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. For this reason modeltranslation adds a thin wrapper (`modeltranslation.utils.get_language`) around the function which guarantees that the returned language is listed in the `LANGUAGES` setting.

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

2.9.2 Using in combination with `django-audit-log`

`django-audit-log` is a package that allows you to track changes to your model instances (documentation). As `django-audit-log` behind the scenes automatically creates “shadow” models for your tracked models, you have to remember to register these shadow models for translation as well as your regular models. Here’s an example:

```python
from modeltranslation.translator import register, TranslationOptions
from my_app import models

@register(models.MyModel)
@register(models.MyModel.audit_log.model)
class MyModelTranslationOptions(TranslationOptions):
    """Translation options for MyModel.""
    fields = ('text', 'title',)
```

If you forget to register the shadow models, you will get an error like:

```
TypeError: 'text_es' is an invalid keyword argument for this function
```

2.9.3 Using in combination with `django-rest-framework`

When creating a new viewset, make sure to override `get_queryset` method, using `queryset` as a property won’t work because it is being evaluated once, before any language was set.
2.10 How to Contribute

There are various ways how you can contribute to the project.

2.10.1 Contributing Code

The preferred way for code contributions are pull requests at Github, usually created against master.

Note: In order to be properly blamed for a contribution, please verify that the email you commit with is connected to your Github account (see help.github.com for details).

Coding Style

Please make sure that your code follows the PEP 8 style guide. The only exception we make is to allow a maximum line length of 100. Furthermore your code has to validate against pyflakes. It is recommended to use flake8 which combines all the checks:

```
$ flake8 --max-line-length=100 modeltranslation
```

The #NOQA mark added by flake8 should be used sparsely.

Django and Python Versions

We always try to support at least the two latest major versions of Django, as well as Django’s development version. While we can not guarantee the latter to be supported in early development stages of a new Django version, we aim to achieve support once it has seen its first release candidate.

The supported Python versions can be derived from the supported Django versions. Example (from the past) where we support Python 2.5, 2.6 and 2.7:

- Django 1.3 (old stable) supports Python 2.5, 2.6, 2.7
- Django 1.4 (current stable) supports Python 2.5, 2.6, 2.7
- Django 1.5 (dev) supports Python 2.6, 2.7

Python 3 is supported since 0.7 release. Although 0.6 release supported Django 1.5 (which started Python 3 compliance), it was not Python 3 ready yet.

Unittests

Modeltranslation has a comprehensive test suite. A test runner is provided which allows to run the tests outside of a Django project:

```
$ python runtests.py
```

Non trivial changes and new features should always be accompanied by a unittest. Pull requests which add unittests for uncovered code or rare edge cases are also appreciated.
Continuous Integration

The project uses Travis CI for continuous integration tests. Hooks provided by Github are active, so that each push and pull request is automatically run against our Travis CI config, checking code against different databases, Python and Django versions. This includes automatic tracking of test coverage through Coveralls.

2.10.2 Contributing Documentation

Documentation is a crucial part of any open source project. We try to make it as useful as possible for both, new and experienced developers. If you feel that something is unclear or lacking, your help to improve it is highly appreciated.

Even if you don’t feel comfortable enough to document modeltranslation’s usage or internals, you still have a chance to contribute. None of the core committers is a native english speaker and bad grammar or misspellings happen. If you find any of these kind or just simple typos, nobody will feel offended for getting an English lesson.

The documentation is written using reStructuredText and Sphinx. You should try to keep a maximum line length of 80 characters. Unlike for code contribution this isn’t a forced rule and easily exceeded by something like a long url.

2.10.3 Using the Issue Tracker

When you have found a bug or want to request a new feature for modeltranslation, please create a ticket using the project’s issue tracker. Your report should include as many details as possible, like a traceback in case you get one.

Please do not use the issue tracker for general questions, we run a dedicated mailing list for this.

2.11 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.

2.11.1 django-multilingual

A library providing support for multilingual content in Django models.

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

2.11.2 django-multilingual-model

A much simpler version of the above django-multilingual.

It works very similar to the django-multilingual approach.
2.11.3 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.

2.11.4 i18ndynamic

This approach is not developed any more.

2.11.5 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.

2.12 ChangeLog
3.1 Core Committers

- Peter Eschler <peschler@gmail.com> (retired)
- Dirk Eschler <eschler@gmail.com>
- Jacek Tomaszewski <jacek.tomek@gmail.com>

3.2 Contributors

- Carl J. Meyer
- Jaap Roes
- Bojan Mihelac
- Sébastien Fievet
- Bruno Tavares
- Zach Mathew (of django-linguo, initial author of MultilingualManager)
- Mihai Sucan
- Benoît Bryon
- Wojtek Ruszczewski
- Chris Adams
- Dominique Lederer
- Braden MacDonald
- Karol Fuksiewicz
- Konrad Wojas
• Bas Peschier
• Oleg Prans
• Francesc Arpí Roca
• Mathieu Leplatre
• Thom Wiggers
• Warnar Boekkooi
• Alex Marandon
• Fabio Caccamo
• Vladimir Sinitsin
• Luca Corti
• Morgan Aubert
• Mathias Ettinger
• Daniel Loeb
• Stephen McDonald
• Lukas Lundgren
• zenoamaro
• oliphunt
• Venelin Stoykov
• Stratos Moros
• Benjamin Toueg
• Emilie Zawadzki
• Virgilio N Santos
• PetrDlouhy
• dmarcelino
• GreyZmeem
• And many more … (if you miss your name here, please let us know!)
empty_values (TranslationOptions attribute), 15

fallback_languages (TranslationOptions attribute), 15
fallback_undefined (TranslationOptions attribute), 15
fallback_values (TranslationOptions attribute), 15
fields (TranslationOptions attribute), 15

required_languages (TranslationOptions attribute), 15