
django-modeltranslation Documentation

Release 0.7.3

Dirk Eschler

Aug 22, 2017

Contents

1	Features	3
1.1	Project Home	3
1.2	Documentation	3
1.3	Mailing List	3
2	Table of Contents	5
2.1	Installation	5
2.2	Setup	6
2.3	Configuration	6
2.4	Registering Models for Translation	10
2.5	Accessing Translated and Translation Fields	13
2.6	ModelForms	19
2.7	Django Admin Integration	20
2.8	Management Commands	25
2.9	Caveats	26
2.10	How to Contribute	27
2.11	Related Projects	28
2.12	ChangeLog	29
3	Authors	39
3.1	Core Committers	39
3.2	Contributors	39

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.

- Add translations without changing existing models
- Fast, because translation fields are stored in the same table
- Supports inherited models
- Django admin support
- Unlimited number of target languages

Project Home

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

Documentation

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

Mailing List

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

Installation

Requirements

Modeltranslation	Python	Django
>=0.7	3.2 - 3.3	1.5 - 1.6
	2.6 - 2.7	1.4 - 1.6
==0.5, ==0.6	2.6 - 2.7	1.5
	2.5 - 2.7	1.3 - 1.4
==0.4	2.5 - 2.7	1.3 - 1.4
<=0.3	2.4 - 2.7	1.0 - 1.4

Using Pip

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

Using the Source

Get a source tarball from [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.

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. 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 `./manage.py syncdb` (note that this only applies if the models registered in the `translation.py` did not have been synced to the database before. If they did - read *further down* what to do in that case.

Configuration

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',  
    ...  
)
```

`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.

Warning: Modeltranslation does not enforce the `LANGUAGES` setting to be defined in your project. When it isn't present, it defaults to Django's global `LANGUAGES` setting instead, and that are quite a number of languages!

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_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:

```
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 similar 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',  
)
```

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 `MODELTRANSLATION_TRANSLATION_REGISTRY` is handled no more.

MODELTRANSLATION_CUSTOM_FIELDS

Default: () (empty tuple)

New in version 0.3.

Modeltranslation supports the fields listed in the *Supported Fields Matrix*. In most cases subclasses of the supported fields will work fine, too. Unsupported fields 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 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.

MODELTRANSLATION_AUTO_POPULATE

Default: False

New in version 0.5.

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:

```
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 --')
```

Possible modes are listed [here](#).

MODELTRANSLATION_DEBUG

Default: False

New in version 0.4.

Changed in version 0.7.

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.

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.

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 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.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:

```
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 auto-magical fields. The next section explains how things are working under the hood.

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).

```
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:

```
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, 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).

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:

```
$ ./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,
)
;
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.

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 third-party app that is using South, things get more complicated. In order to be able to update the app in future, and to feel comfortable, you should use the `sync_translation_fields` 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 declared `blank=True` and `null=True` no matter if the original field is required or not. In other words - all translations are optional. To populate the default translation fields added by the modeltranslation application with values from existing database fields, you can use the `update_translation_fields` command below. See *The `update_translation_fields` Command* for more info on this.

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*.

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*.

```
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*.

```
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*.

```
fallback_undefined = None
fallback_undefined = {'title': 'no title', 'text': None}
```

`TranslationOptions`. **empty_values**

Override the value that should be saved in forms on empty fields. See *Formfields and nullability*.

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

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:

Model Field	0.4	0.5	0.7
AutoField	No	No	No
BigIntegerField	No	Yes*	Yes*
BooleanField	No	Yes	Yes
CharField	Yes	Yes	Yes
CommaSeparatedIntegerField	No	Yes	Yes
DateField	No	Yes	Yes
DateTimeField	No	Yes	Yes
DecimalField	No	Yes	Yes
EmailField	Yes*	Yes*	Yes*
FileField	Yes	Yes	Yes
FilePathField	Yes*	Yes*	Yes*
FloatField	No	Yes	Yes
ImageField	Yes	Yes	Yes
IntegerField	No	Yes	Yes
IPAddressField	No	Yes	Yes
GenericIPAddressField	No	Yes	Yes
NullBooleanField	No	Yes	Yes
PositiveIntegerField	No	Yes*	Yes*
PositiveSmallIntegerField	No	Yes*	Yes*
SlugField	Yes*	Yes*	Yes*
SmallIntegerField	No	Yes*	Yes*
TextField	Yes	Yes	Yes
TimeField	No	Yes	Yes
URLField	Yes*	Yes*	Yes*
ForeignKey	No	No	Yes
OneToOneField	No	No	Yes
ManyToManyField	No	No	No

* Implicitly supported (as subclass of a supported field)

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*.

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*.

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.

Todo

Add more examples.

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:

```
# 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()`
- `dates()`
- `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:

```
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:

```
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 4 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

Falling back

Modeltranslation provides mechanism to control behaviour of data access in case of empty translation values. This mechanism affects field access.

Consider `News` example: a creator of some news hasn't specified it's german title and content, but only english ones. Then if a german visitor is viewing site, we would rather show him english title/content of the news than display empty strings. This is called *fallback*.

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

There are several ways of controlling fallback, described below.

Fallback languages

New in version 0.5.

`MODELTRANSLATION_FALLBACK_LANGUAGES` setting allows to set order of *fallback languages*. By default it is only `DEFAULT_LANGUAGE`.

For example, setting

```
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 yield non-empty value of the field.

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

```
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* (Ukrainian) order of fallback languages is: ('ru', 'en', 'de', 'fr')
- for *fr* order of fallback languages is: ('de', 'en') - *fr* obviously is not fallback, since it's active language; and *de* would be tried before *en*
- for *en* and *de* fallback order is ('de', 'fr') and ('en', 'fr'), respectively
- for any other language 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.

New in version 0.6.

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
```

Fallback values

New in version 0.4.

But what if current language and all fallback languages yield no field value? Then `modeltranslation` will use 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 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 value.

`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 fallback 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
    }
```

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.

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.

ModelForms

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:

```
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.

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 field is nullable (see [django ticket #9590](#)).

Thus formfields for translation fields are patched by *MT*. Following rules apply:

- If original field is not nullable, empty value would be saved as '';

- If original field is nullable, empty value would be saved as `None`.

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

```
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:

```
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 error when 2 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 database don't violate uniqueness:

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

None-checkbox widget

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

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

It results in 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. Probably only useful place for this widget is admin backend; see *Formfields with None-checkbox*.

To sum up, only valid `empty_values` values are: `None`, `' '` and `'both'`.

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)
```

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 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/_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 - 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:

```
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',)})]
```

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.

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 a new model called `Image`. The 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)

```

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',),
        }
```

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:

```
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:

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

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.

```
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')
```

Management Commands

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 (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:

```
$ ./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 default translation field `title_de`. It only does so if the default translation field is empty otherwise nothing is copied.

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.

The `sync_translation_fields` Command

New in version 0.4.

```
$ ./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 new language or deciding to translate new field in a `TranslationOptions`.

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.

The `loaddata` Command

New in version 0.7.

It is just extension to original `loaddata` command which adds an optional `populate` keyword. If specified, then normal loading command will be run under selected auto-population modes.

By default no auto-population is performed.

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

Allowed modes are listed [here](#). To choose `False` (turn off auto-population) specify `'0'` or `'false'`:

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

Note: If `populate` is not specified, then 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 proper locale. To get back to old behaviour, set `MODELTRANSLATION_LOADDATA_RETAIN_LOCALE` to `False`.

Caveats

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 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.

How to Contribute

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

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 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 currently not supported, but should be added no later than it becomes officially supported by Django.

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](#).

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.

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.

Related Projects

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

django-multilingual

A library providing support for multilingual content in Django models.

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

django-multilingual-model

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

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

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.

i18ndynamic

This approach is not developed any more.

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.

ChangeLog

v0.7.3

=====

Date: 2014-01-05

- ADDED: Documentation for TranslationOptions fields reference and south/sync_translation_fields.
- FIXED: Some python3 compatibility issues.
(thanks Jacek Tomaszewski,
resolves issue #220)
- FIXED: Clearing translated FileFields does not work with easy_thumbnails.
(thanks Jacek Tomaszewski,
resolves issue #219)
- FIXED: Compatibility with nested inlines.
(thanks abstraktor,
resolves issue #218)
- FIXED: Admin inlines recursion problem in Django 1.6.
(thanks Oleg Prans,
resolves issue #214)
- FIXED: Empty FileField handling.
(thanks Jacek Tomaszewski,
resolves issue #215)

v0.7.2

=====

Date: 2013-11-11

- ADDED: Documentation about empty_values.
(thanks Jacek Tomaszewski,
resolves issue #211)
- FIXED: Proxy model handling.
(thanks Jacek Tomaszewsk)
- FIXED: Abstract managers patching.
(thanks Jacek Tomaszewski,
resolves issue #212)

v0.7.1

=====

Date: 2013-11-07

Packaged from revision f7c7ea174344f3dc0cf56ac3bf6e92878ed6baea

ADDED: Configurable formfields. The ultimate approach to nullable CharFields.
(thanks Jacek Tomaszewski,
resolves issue #211, ref #163, #187)

FIXED: Recursion problem with fieldset handling in Django 1.6.
(thanks to Bas Peschier,
resolves issue #214)

v0.7

====

Date: 2013-10-19

Packaged from revision 89f5e6712aaf5d5ec7e2d61940dcla71fb08ca94

ADDED: A setting to control which language are slug fields based on
(thanks to Konrad Wojas,
resolves issue #194)

ADDED: A noinput option to the sync_translation_fields management command.
(thanks to cuchac,
resolves issues #179 and #184)

ADDED: Support for Python 3.2 and 3.3.
(thanks to Karol Fuksiewicz,
resolves issue #174)

ADDED: Convenient admin classes which already contain proper Media
definitions.
(resolves issue #171)

ADDED: Only, defer, values, values_list, dates, raw_values methods to
MultilingualManager.
(resolves issue #166 adn #173)

ADDED: Support for ForeignKey and OneToOneField.
(thanks to Braden MacDonald and Jacek Tomaszewski,
resolves issue #161)

ADDED: An auto-population option to the loaddata command.
(resolves issue #160)

ADDED: A MODELTRANSLATION_LOADDATA_RETAIN_LOCALE setting for loaddata
command to leave locale alone.
(resolves issue #151)

FIXED: Compatibility with Django 1.6 development version.
(resolves issue #169)

FIXED: Handling of 3rd party apps' ModelForms.
(resolves issue #167)

FIXED: Triggering field fallback on its default value rather than empty
string only. Also enhance nullable fields in forms with proper
widgets to preserve ``None``.
(thanks to Wojtek Ruszczewski,
resolves issue #163)

FIXED: Admin prepopulated_fields is now handled properly.
(thanks to Rafleze,
resolves issue #181 and #190)

FIXED: Form saving when translated field is excluded (e.g. in admin)
(resolves issue #183)

FIXED: Multilingual clones are Multilingual too.
(resolved issue #189)

CHANGED: Every model's manager is patched as MultiLingual, not only objects.
 (resolved issue #198)
 CHANGED: Display "make null" checkboxes in model forms.
 CHANGED: MODELTRANSLATION_DEBUG setting defaults to False instead of
 settings.DEBUG.
 CHANGED: Drop support for Python 2.5 and Django 1.3.

v0.6.1

=====

Date: 2013-03-17

Packaged from revision fc8a3034897b8b818c74f41c43a92001e536d970

FIXED: Joined query does not use translated fields.
 (resolves issue #162)

v0.6

=====

Date: 2013-03-01

Packaged from revision ea0e2db68900371146d39dcdf88b29091ee5222f

ADDED: A new ENABLE_FALLBACKS setting and a context manager for switching
 fallbacks temporarily.
 (thanks to Wojtek Ruszczewski,
 resolves issue #152)
 ADDED: Major refactoring of the tabbed translation fields javascript. Adds
 support for tabular inlines and includes proper handling of stacked
 inlines, which have never been officially supported, but were not
 actively prevented from being tabbified.
 (resolves issue #66)
 ADDED: New group_fieldsets option for TranslationAdmin. When activated
 translation fields and untranslated fields are automatically
 grouped into fieldsets.
 (based on original implementation by Chris Adams,
 resolves issues #38)
 FIXED: Tests to run properly in the scope of a Django project.
 (thanks to Wojtek Ruszczewski,
 resolves issue #153)
 FIXED: Broken tab activation when using jquery-ui 1.10, keeping support for
 older jquery-ui versions and the jquery version shipped by Django.
 (thanks to Dominique Lederer,
 resolves issue #146)
 FIXED: Wrong admin field css class for en-us language.
 (resolves issue #141)
 FIXED: Added missing hook for admin readonly_fields.
 (resolves issue #140)
 FIXED: Keys used in tabbed translation fields to group translations are not
 unique for inlines.
 (resolves issue #121)
 FIXED: The prepopulated_fields TranslationAdmin option only works on the
 first defined field to prepopulate from and made the option aware
 of the current language.
 (resolves issue #57)

CHANGED: Removed deprecated MODELTRANSLATION_TRANSLATION_REGISTRY setting.

CHANGED: Refactored auto population manager functionality. Switched to a

```
    populate method in favour of the old _populate keyword and added a new
    contextmanager to switch the population mode on demand.
    (thanks to Wojtek Ruszczewski,
     resolves issue #145)
CHANGED: Major refactoring of translation field inheritance and
    TranslationOptions.
    (thanks to Wojtek Ruszczewski,
     resolves issues #50 and #136)
```

v0.5

====

Date: 2013-02-10

Packaged from revision bedd18ea9e338b133d06f2ed5e7ebfc2e21fd276

```
ADDED: Merged autodiscover tests from django-modeltranslation-wrapper.
ADDED: Rewrite method to MultilingualManager and optimized create.

FIXED: grouped_translations are computed twice in tabbed translations.
    (thanks to Wojtek Ruszczewski,
     resolves issue #135)
FIXED: CSS classes in tabbed translation fields when fieldname has a leading
    underscore.
    (thanks to Wojtek Ruszczewski,
     resolves issue #134)
FIXED: Rewriting of descending ('-' prefixed) ordering fields in
    MultilingualManager.
    (thanks to Wojtek Ruszczewski,
     resolves issue #133)
FIXED: Download url in setup.py.
    (thanks to Benoît Bryon,
     resolves issue #130)
FIXED: The update_translation_fields management command does nothing.
    (resolves issue #123)
FIXED: MultilingualQuerySet custom inheritance.

CHANGED: Don't raise an exception if TranslationField is accessed via class
    to allow descriptor introspection.
    (resolves issue #131)
```

v0.5b1

=====

Date: 2013-01-07

Packaged from revision da928dd431fcf112e2e9c4c154c5b69e7dad3b3.

```
ADDED: Possibility to turn off query rewriting in MultilingualManager.
    (thanks to Jacek Tomaszewski)

FIXED: Fixed update_translation_fields management command.
    (thanks to Jacek Tomaszewski,
     resolves issues #123 and #124)

CHANGED: Major test refactoring.
    (thanks to Jacek Tomaszewski,
     resolves issues #100 and #119)
```

v0.5a1

=====

Date: 2012-12-05

Packaged from revision da4aeba0ea20ddb6e67aa49bc90af507997ac386.

- ADDED: Increased the number of supported fields. Essentially all Django model fields and subclasses of them should work, except related fields (ForeignKey, ManyToManyField, OneToOneField) and AutoField which are not supported.
- ADDED: A subclass of TranslationOptions inherits fields from its bases. (thanks to Bruno Tavares and Jacek Tomaszewski, resolves issue #110)
- ADDED: Support for fallback languages. Allows fine grained configuration through project settings and TranslationOptions on model basis. (thanks to Jacek Tomaszewski, resolves issue #104)
- ADDED: Multilingual manager which is aware of the current language. (thanks to Jacek Tomaszewski, resolves issues #45, #78 and #84)
- CHANGED: Version code to use a PEP386 compliant version number.
- CHANGED: Constructor rewrites fields to be language aware. (thanks to Jacek Tomaszewski, resolves issues #33 and #58)
- FIXED: Lacking support for readonly_fields in TranslationAdmin. (thanks to sbrandtb, resolves issue #111)
- FIXED: Model's db_column option is not applied to the translation field. (resolves issue #83)
- FIXED: Admin prevents saving a cleared field. The fix deactivates rule3 and implies the new language aware manager and constructor rewrite. (resolves issue #85)

v0.4.1

=====

Date: 2012-11-13

Packaged from revision d9bf9709e9647fb2af51fc559bbe356010bd51ca.

- FIXED: Pypi wants to install beta version. Happened because pypi treats 0.4.0-beta2 as latest release. This also effectively resulted in a downgrade when using 'pip --upgrade' and 0.4.0 was already installed. (thanks to jmagnusson for the report, resolves issue #103)

v0.4.0

=====

Date: 2012-11-11

Packaged from revision c44f9cfee59f1b440f022422f917f247e16bbc6b.

- CHANGED: Refactored tests to allow test runs with other apps. Includes a "backport" of override_settings to ensure Django 1.3 support. (thanks to Jacek Tomaszewski)
- CHANGED: Modeltranslation related css class prefix to 'mt'.
- FIXED: Race condition during initialization. (resolves issue #91)

FIXED: Tabs don't properly support two-part language codes.
(resolves issue #63)

v0.4.0-beta2

=====

Date: 2012-10-17

Packaged from revision 7b8cafbde7b14afc8e85235e9b087889a6bfa86e.

FIXED: Release doesn't include rst files.

v0.4.0-beta1

=====

Date: 2012-10-17

Packaged from revision 09a0c4434a676c6fd753e6dcde95056c424db62e.

CHANGED: Refactored documentation using sphinx.
(resolves issue #81)

FIXED: Setting MODELTRANSLATION_TRANSLATION_FILES should be optional.
(resolves issue #86)

v0.4.0-alpha1

=====

Date: 2012-10-12

Packaged from revision 170.

ADDED: Support for FileField and ImageField.
(thanks to Bruno Tavares,
resolves issue #30)

ADDED: New management command sync_database_fields to sync the database after
a new model has been registered or a new language has been added.
(thanks to Sébastien Fievet and the authors of django-transmeta,
resolves issue #62)

CHANGED: Excluded tabular inlines from jQuery tabs, as they are currently
not supported.

CHANGED: Use app-level translation files in favour of a single project-level
one. Adds an autoregister feature similiar to the one provided by
Django's admin. A new setting MODELTRANSLATION_TRANSLATION_FILES keeps
backwards compatibility with older versions. See documentation for
details. This is basically a merge from both
django-modeltranslation-wrapper and hyperweek's branch at github.
(thanks to Jacek Tomaszewski, Sébastien Fievet and Maxime Haineault,
resolves issues #19, #58 and #71)

CHANGED: Moved tests to separate folder and added tests for TranslationAdmin.
To run the tests the settings provided in model.tests.modeltranslation
have to be used (settings.LANGUAGES override doesn't work for
TranslationAdmin).

CHANGED: Major refactoring of the admin integration. Subclassed BaseModelAdmin
and InlineModelAdmin. Patching options in init doesn't seem to be
thread safe. Instead used provided hooks like get_form, get_formset
and get_fieldsets. This should resolve several problems with the
exclude and fieldsets options and properly support options in inlines.
(resolves issue #72)

FIXED: Non-unicode verbose field names showing up empty in forms.
(resolves issue #35)

FIXED: Dynamic TranslationOptions model name.

FIXED: Widgets for translated fields are not properly copied from original fields.
(thanks to boris-chervenkov, resolves issue #74)

FIXED: Removed XMLField test which is deprecated since Django 1.3 and broke tests in Django 1.4.
(resolves issue #75)

v0.3.3

=====

Date: 2012-02-23

Packaged from revision 129.

CHANGED: jQuery search path in `tabbed_translation_fields.js`. This allows use of a version of jQuery other than the one provided by Django. Users who want to force the use of Django's jQuery can include `force_jquery.js`.

FIXED: Another attempt to include static files during installation.
(resolves reopened issue #61)

v0.3.2

=====

Date: 2011-06-16

Packaged from revision 122.

FIXED: Static files not included during installation.
(resolves issue #61)

v0.3.1

=====

Date: 2011-06-07

Packaged from revision 121.

CHANGED: Renamed media folder to static.

v0.3

=====

Date: 2011-06-03

Packaged from revision 113.

ADDED: Support for multi-table inheritance.
(thanks to Sébastien Fievet, resolves issues #50 and #51)

ADDED: JQuery-ui based admin support for tabbed translation fields.
(thanks to jaap and adams, resolves issue #39)

ADDED: CSS class to identify a translation field and the default translation field in admin.
(thanks to jaap)

ADDED: Configurable default value per field instance.
(thanks to bmihelac, resolves issue #28)

ADDED: Setting to override the default language.
(thanks to jaap, resolves issue #2)

CHANGED: Improved performance of update_translation_fields command.
(thanks to adamsc, resolves issue #43)

CHANGED: Factored out settings into a separate settings.py and consistently used an app specific settings prefix.

CHANGED: Refactored creation of translation fields and added handling of supported fields.
(resolves issue #37)

FIXED: Clearing the default translation field in admin does not clear the original field.
(resolves issue #47)

FIXED: In some setups appears "This field is required" error for the original field.
(resolves issue #5)

FIXED: Translations are not saved for tinymce HTMLField when using jquery tabs.
(thanks to kottenator, resolves issue #41)

FIXED: Fieldname isn't ensured to be string.
(resolves issue #41)

FIXED: Kept backwards compatibility with Django-1.0.
(thanks to jaap, resolves issue #34)

FIXED: Regression in south_field_triple caused by r55.
(thanks to jaap, resolves issue #29)

FIXED: TranslationField pre_save does not get the default language correctly.
(thanks to jaap, resolves issue #31)

v0.2

====

Date: 2010-06-15

Packaged from revision 57.

ADDED: Support for admin prepopulated_fields.
(resolves issue #21)

ADDED: Support for admin list_editable.
(thanks carl.j.meyer, resolves issue #20)

ADDED: Preserve the formfield widget of the translated field.
(thanks piquadrat)

ADDED: Initial support for django-south.
(thanks andrewgodwin, resolves issue #11)

ADDED: Support for admin inlines, common and generic.
(resolves issue #12 and issue #18)

FIXED: Admin form validation errors with empty translated values and unique=True.
(thanks to adamsc, resolves issue #26)

FIXED: Mangling of untranslated prepopulated fields.
(thanks to carl.j.meyer, resolves issue #25)

FIXED: Verbose names of translated fields are not translated.
(thanks to carl.j.meyer, resolves issue #24)

FIXED: Race condition between model import and translation registration in production by ensuring that models are registered for translation before TranslationAdmin runs.
(thanks to carl.j.meyer, resolves issue #19)

FIXED: Added workaround for swallowed ImportErrors by printing a traceback explicitly.
(resolves issue #17)

```
FIXED: Only print debug statements to stdout if the runserver or
runserver_plus management commands are used.
(resolves issue #16)
FIXED: Removed print statements so that modeltranslation is usable with
mod_wsgi.
(resolves issue #7)
FIXED: Broken admin fields and fieldsets.
(thanks simoncelen, resolves issue #9)
FIXED: Creation of db fields with invalid python language code.
(resolves issue #4)
FIXED: Tests to run from any project.
(thanks carl.j.meyer, resolves issue #6)
FIXED: Removed unused dependency to content type which can break syncdb.
(thanks carl.j.meyer, resolves issue #1)
```

v0.1

====

Date: 2009-02-22

Initial release packaged from revision 19.

Core Committers

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

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
- And many more ... (if you miss your name here, please let us know!)

E

`empty_values` (TranslationOptions attribute), 12

F

`fallback_languages` (TranslationOptions attribute), 12

`fallback_undefined` (TranslationOptions attribute), 12

`fallback_values` (TranslationOptions attribute), 12

`fields` (TranslationOptions attribute), 12