django-i18nfield Documentation

Release 1.9

Raphael Michel

Dec 16, 2021

Contents

1 Documentation content **3**
- 1.1 Getting started . . . 3
- 1.2 Working with translated strings . . . 4
- 1.3 Working with forms . . . 5
- 1.4 Admin integration . . . 8

Index **9**

This is yet another way to store multi-lingual content in Django. In contrast to other options like django-havd, django-modeltranslation or django-parler it does not require additonal database tables and you can reconfigure the available languages without any changes to the database schema. In constrast to nece, it is not specific to PostgreSQL.

How does it work then? It stores JSON data into a `TextField`. Yes, this is kinda dirty and violates the 1NF. This makes it harder for non-django based programs to interact directly with your database and is not perfectly efficient in terms of storage space. It also lacks the ability to do useful lookups, searches and indices on internationalized fields. If one of those things are important to you, **this project is not for you**, please choose one of the ones that we linked above.

However if those limitations are fine for you, this provides you with a very lightweight, easy to use and flexible solution. This approach has been in use in pretix for quite a while, so it has been tested in production. The package contains not only the model fields, but also form fields and everything you need to get them running.

CHAPTER 1

Documentation content

1.1 Getting started

First of all, you need to install django-i18nfield:

```
$ pip3 install django-i18nfield
```

You should also check that your `settings.py` lists the languages that you want to use:

```
from django.utils.translation import gettext_lazy as _

LANGUAGES = [
    ('de', _('German')),
    ('en', _('English')),
    ('fr', _('French')),
]
```

Now, let's assume you have a simple django model like the following:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(verbose_name='Book title', max_length=190)
    abstract = models.TextField(verbose_name='Abstract')
    author = models.ForeignKey(Author, verbose_name='Author')
```

You can change your model to store internationalized data like the following:

```
from django.db import models
from i18nfield.fields import I18nCharField, I18nTextField

class Book(models.Model):
    title = I18nCharField(verbose_name='Book title', max_length=190)
```

(continues on next page)

(continued from previous page)

```
    abstract = I18nTextField(verbose_name='Abstract')
    author = models.ForeignKey(Author, verbose_name='Author')
```

Then, create a migration as you would for any database change:

```
$ python manage.py makemigrations
```

And you're done! Really, that's it.

If you now create a `ModelForm` for that model, the title and author fields will consist of multiple language fields, one for each language. They don't look nice yet and Django admin does not know how to deal with them so far. Also, they no longer contain standard python strings but `LazyI18nStrings` which have some special property. But luckily for you, we wrote more pages in this documentation, go ahead and check them out. :)

1.2 Working with translated strings

If you want to save multi-lingual data into an `I18nCharField` or `I18nTextField`, you need to wrap it as an `LazyI18nString` first. Also, if you read data from such a field, you will always get an `LazyI18nString` back.

An `LazyI18nString` is a representation of a string that exists in multiple languages. Under the hood, it is just a dictionary that maps languages to values.

So why don't we just use dictionaries? This is where the "**lazy**" comes into play: As soon as you try to evaluate an `LazyI18nString` as a normal string, it will magically transform into a normal string – based on the currently active language. This means that e.g. if you get a value from an `I18nCharField` and pass it to a template, the template will cast the value to a string and you **do not need to do anything** to make it work.

This behaviour is intentionally very similar to the `gettext_lazy` method from Django's translation layer.

However, when you deal with such strings in python code, you should know how they behave. Therefore, we have a number of examples for you on this page.

To create a LazyI18nString, we can input a simple string:

```
>>> naive = LazyI18nString('Naive untranslated string')
>>> naive
<LazyI18nString: 'Naive untranslated string'>
```

Or we can provide a dictionary with multiple translations:

```
>>> translated = LazyI18nString(
...     {'en': 'English String', 'de': 'Deutscher String'}
... )
```

We can use the `localize` method to get the string in a specific language:

```
>>> translated.localize('de')
'Deutscher String'

>>> translated.localize('en')
'English String'
```

If we try a locale that does not exist for the string, we might get a it either in a similar locale or in the system's default language:

```
>>> translated.localize('de-AT')
'Deutscher String'

>>> translated.localize('zh')
'English String'

>>> naive.localize('de')
'Naive untranslated string'
```

Important: This is an important property of LazyI18nString: **As long as there is any non-empty value for any language, you will rather get a result in the wrong language than an empty result.** This makes it "safe" to use if your data is only partially translated.

If we cast a `LazyI18nString` to `str`, `localize` will be called with the currently active language:

```
>>> from django.utils import translation
>>> str(translated)
'English String'
>>> translation.activate('de')
>>> str(translated)
'Deutscher String'
```

Formatting also works as expected:

```
>>> translation.activate('de')
>>> '{}'.format(translated)
'Deutscher String'
```

If we want to modify all translations inside a `LazyI18nString` we can do so using the `map` method:

```
>>> translated.map(lambda s: s.replace('String','Text'))
>>> translation.activate('de')
>>> str(translated)
'Deutscher Text'
```

There is also a way to construct a hybrid object that takes its data from `gettext` but behaves like an `LazyI18nString`. The use case for this is very rare, it basically only is useful when defining default values for internationalized form fields in the codebase.

```
>>> from django.utils.translation import gettext_noop
>>> LazyI18nString.from_gettext(gettext_noop('Hello'))
<LazyI18nString: <LazyGettextProxy: 'Hello'>>
```

1.3 Working with forms

1.3.1 Fields and Widgets

If you use a `ModelForm`, you will automatically get an `I18nFormField` field for your internationalized fields with the default widget being either an `I18nTextInput` or an `I18nTextarea` being the default widget. But of course you can also use these fields manually as you would use any other field, even completely without touching models.

class i18nfield.forms.**I18nFormField**(**args*, ***kwargs*)

The form field that is used by I18nCharField and I18nTextField. It makes use of Django's MultiValueField mechanism to create one sub-field per available language.

It contains special treatment to make sure that a field marked as "required" is validated as "filled out correctly" if *at least one* translation is filled it. It is never required to fill in all of them. This has the drawback that the HTML property `required` is set on none of the fields as this would lead to irritating behaviour.

Parameters

- **locales** – An iterable of locale codes that the widget should render a field for. If omitted, fields will be rendered for all languages configured in `settings.LANGUAGES`.
- **require_all_fields** – A boolean, if set to True field requires all translations to be given.

class i18nfield.forms.**I18nTextInput**(*locales: List[str]*, *field: django.forms.fields.Field*, *attrs=None*)

The default form widget for I18nCharField. It makes use of Django's MultiWidget mechanism and does some magic to save you time.

class i18nfield.forms.**I18nTextarea**(*locales: List[str]*, *field: django.forms.fields.Field*, *attrs=None*)

The default form widget for I18nTextField. It makes use of Django's MultiWidget mechanism and does some magic to save you time.

1.3.2 Widget styling

The form widget will output something similar to the following HTML sample:

```
<div class="i18n-form-group">
    <input class="form-control" id="id_name_0" lang="en"
           maxlength="200" name="name_0" placeholder="Name" title="en"
           type="text" value="">
    <input class="form-control" id="id_name_1" lang="de"
           maxlength="200" name="name_1" placeholder="Name" title="de"
           type="text" value="">
</div>
```

This alone provides no good indication to your user on which field resembles which language (except the title attribute that is visible on mouseover in most browsers). Also, it will render the input forms in a row by default, why we find it more understandable if they are arranged vertically.

You can achieve all this with a little bit of CSS. We can't give you the full details, as we don't know how you style form widgets in general in your project.

To indicate the language, we use the following CSS to draw a little flag at the beginning of the input field:

```
input[lang] {
  background: no-repeat 10px center;
  padding-left: 34px;
}
textarea[lang] {
  background: no-repeat 10px 10px;
  padding-left: 34px;
}
input[lang=de], textarea[lang=de] {
  background-image: url('/static/img/flags/de.png');
}
```

(continues on next page)

(continued from previous page)

```
input[lang=en], textarea[lang=en] {
  background-image: url('/static/img/flags/en.png');
}
```

In pretix, this looks like this:

1.3.3 Advanced usage: Restrict the visible languages

Sometimes, you do not want to display fields for all languages every time. If you build a shopping platform, your platform might support tens or hundreds of languages, while a single shop only supports a few of them. In this case, the shop owner should not see input fields for languages that they don't want to support.

As you can see above, `I18nFormField` has a constructor argument `locales` that takes a list of locales for this exact purpose. However, most of the time, your `I18nFormField` is defined in a way that does not allow you to pass a dynamic list there. Therefore, we provide a form base class that you can use for your `ModelForm` that *also* takes a `locales` constructor argument and passes it through to all its fields.

For the same reason, we provide formset base classes that add the `locales` argument to your formset class and pass it through to all fields.

class i18nfield.forms.**I18nForm**(**args*, ***kwargs*)

This is a modified version of Django's Form which differs from Form in only one way: The constructor takes one additional optional argument `locales` expecting a list of language codes. If given, this instance is used to select the visible languages in all I18nFormFields of the form. If not given, all languages from `settings.LANGUAGES` will be displayed.

Parameters **locales** – A list of locales that should be displayed.

class i18nfield.forms.**I18nModelForm**(**args*, ***kwargs*)

This is a modified version of Django's ModelForm which differs from ModelForm in only one way: The constructor takes one additional optional argument `locales` expecting a list of language codes. If given, this instance is used to select the visible languages in all I18nFormFields of the form. If not given, all languages from `settings.LANGUAGES` will be displayed.

Parameters **locales** – A list of locales that should be displayed.

class i18nfield.forms.**I18nModelFormSet**(**args*, ***kwargs*)

This is equivalent to a normal BaseModelFormset, but cares for the special needs of I18nForms (see there for more information).

Parameters **locales** – A list of locales that should be displayed.

class i18nfield.forms.**I18nInlineFormSet**(**args*, ***kwargs*)

This is equivalent to a normal BaseInlineFormset, but cares for the special needs of I18nForms (see there for more information).

Parameters **locales** – A list of locales that should be displayed.

Note: As `I18nFormField` tries to pass this information down to the widget, this might fail if you use a custom widget class that does not inherit from our default widgets.

1.4 Admin integration

Currently, our fields do not yet automatically integrate with Django admin. By default, Django admin tries to use a custom widget for the fields which do not work in our case.

There is a workaround by using a custom admin class, which explicitly defines the widget classes in the background.

```
from i18nfield.admin import I18nModelAdmin

class BookAdmin(I18nModelAdmin):
    pass

admin.site.register(Book, BookAdmin)
```

Index

I

I18nForm (*class in i18nfield.forms*), 7
I18nFormField (*class in i18nfield.forms*), 5
I18nInlineFormSet (*class in i18nfield.forms*), 7
I18nModelForm (*class in i18nfield.forms*), 7
I18nModelFormSet (*class in i18nfield.forms*), 7
I18nTextarea (*class in i18nfield.forms*), 6
I18nTextInput (*class in i18nfield.forms*), 6