Django

Documentation

Translation

Overview

In order to make a Django project translatable, you have to add a minimal number of hooks to your Python code and templates. These hooks are called translation strings. They tell Django: "This text should be translated into the end user's language, if a translation for this text is available in that language." It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

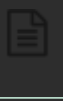
Django then provides utilities to extract the translation strings into a message file. This file is a convenient way for translators to provide the equivalent of the translation strings in the target language. Once the translators have filled in the message file, it must be compiled. This process relies on the GNU gettext toolset.

Once this is done, Django takes care of translating Web apps on the fly in each available language, according to users' language preferences.

Django's internationalization hooks are on by default, and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to set `USE_I18N = False` in your settings file. Then Django will make some optimizations so as not to load the internationalization machinery.

Note

There is also an independent but related `USE_L10N` setting that controls if Django should implement format localization. See Format localization for more details.



Note

Make sure you've activated translation for your project (the fastest way is to check if `MIDDLEWARE` includes `django.middleware.locale.LocaleMiddleware`). If you haven't yet, see How Django discovers language preference.

Internationalization: in Python code

Standard translation

Specify a translation string by using the function `gettext()`. It's convention to import this as a shorter alias, `_`, to save typing.

Note

Python's standard library `gettext` module installs `_()` into the global namespace, as an alias for `gettext()`. In Django, we have chosen not to follow this practice, for a couple of reasons:

1. Sometimes, you should use `gettext_lazy()` as the default translation method for a particular file. Without `_()` in the global namespace, the developer has to think about which is the most appropriate translation function.
2. The underscore character (`_`) is used to represent "the previous result" in Python's interactive shell and doctest tests. Installing a global `_()` function causes interference. Explicitly importing `gettext()` as `_()` avoids this problem.

What functions may be aliased as _?

Because of how `xgettext` (used by `makemessages`) works, only functions that take a single string argument can be imported as `_`:

- `gettext()`
- `gettext_lazy()`

In this example, the text `"Welcome to my site."` is marked as a translation string:

```
from django.http import HttpResponse
from django.utils.translation import gettext as _

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

You could code this without using the alias. This example is identical to the previous one:

```
from django.http import HttpResponse
from django.utils.translation import gettext

def my_view(request):
    output = gettext("Welcome to my site.")
    return HttpResponse(output)
```

Translation works on computed values. This example is identical to the previous two:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

Translation works on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(The caveat with using variables or computed values, as in the previous two examples, is that Django's translation-string-detecting utility, **django-admin makemessages**, won't be able to find these strings. More on **makemessages** later.)

The strings you pass to **_()** or **gettext()** can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be **"Today is November 26."**, while a Spanish translation may be **"Hoy es 26 de noviembre."** – with the month and the day placeholders swapped.

For this reason, you should use named-string interpolation (e.g., **%(day)s**) instead of positional interpolation (e.g., **%s** or **%d**) whenever you have more than a single parameter. If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Since string extraction is done by the **xgettext** command, only syntaxes supported by **gettext** are supported by Django. In particular, Python f-strings are not yet supported by **xgettext**, and JavaScript template strings need **gettext** 0.21+.

Comments for translators

If you would like to give translators hints about a translatable string, you can add a comment prefixed with the **Translators** keyword on the line preceding the string, e.g.:

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = gettext("Welcome to my site.")
```

The comment will then appear in the resulting **.po** file associated with the translatable construct located below it and should also be displayed by most translation tools.

Note

Just for completeness, this is the corresponding fragment of the resulting **.po** file:

```
#. Translators: This message appears on the home page only
# path/to/python/file.py:123
msgid "Welcome to my site."
msgstr ""
```

This also works in templates. See Comments for translators in templates for more details.

Marking strings as no-op

Use the function **django.utils.translation.gettext_noop()** to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users – such as strings in a database – but should be translated at the last possible point in time, such as when the string is presented to the user.

Pluralization

Use the function **django.utils.translation.ngettext()** to specify pluralized messages.

ngettext() takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when you need your Django application to be localizable to languages where the number and complexity of plural forms is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where **count** is different from one, irrespective of its value.)

For example:

```
from django.http import HttpResponse
from django.utils.translation import ngettext

def hello_world(request, count):
    page = ngettext(
        'there is %(count)d object',
        'there are %(count)d objects',
        count,
    ) % {
        'count': count,
    }
    return HttpResponse(page)
```

In this example the number of objects is passed to the translation languages as the **count** variable.

Note that pluralization is complicated and works differently in each language. Comparing **count** to 1 isn't always the correct rule. This code looks sophisticated, but will produce incorrect results for some languages:

```
from django.utils.translation import ngettext
from myapp.models import Report

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ngettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(name)s available.',
    count,
) % {
    'count': count,
    'name': name
}
```

Don't try to implement your own singular-or-plural logic; it won't be correct. In a case like this, consider something like the following:

```
text = ngettext(
    'There is %(count)d %(name)s object available.',
    'There are %(count)d %(name)s objects available.',
    count,
) % {
    'count': count,
    'name': Report._meta.verbose_name,
}
```

Note

When using **ngettext()**, make sure you use a single name for every extrapolated variable included in the literal. In the examples above, note how we used the **name** Python variable in both translation strings. This example, besides being incorrect in some languages as noted above, would fail:

```
text = ngettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(plural_name)s available.',
    count,
) % {
    'count': Report.objects.count(),
    'name': Report._meta.verbose_name,
    'plural_name': Report._meta.verbose_name_plural,
}
```

You would get an error when running **django-admin compilemessages**:

```
a format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid'
```

Contextual markers

Sometimes words have several meanings, such as **"May"** in English, which refers to a month name and to a verb. To enable translators to translate these words correctly in different contexts, you can use the **django.utils.translation.pgettext()** function, or the **django.utils.translation.npgettext()** function if the string needs pluralization. Both take a context string as the first variable.

In the resulting **.po** file, the string will then appear as often as there are different contextual markers for the same string (the context will appear on the **msgctxt** line), allowing the translator to give a different translation for each of them.

For example:

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

or:

```
from django.db import models
from django.utils.translation import pgettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=pgettext_lazy(
        'help text for MyThing model', 'This is the help text'))
```

will appear in the **.po** file as:

```
msgctxt "month name"
msgid "May"
msgstr ""
```

Contextual markers are also supported by the **translate** and **blocktranslate** template tags.

Lazy translation

Use the lazy versions of translation functions in **django.utils.translation** (easily recognizable by the **lazy** suffix in their names) to translate strings lazily – when the value is accessed rather than when they're called.

These functions store a lazy reference to the string – not the actual translation. The translation itself will be done when the string is used in a string context, such as in template rendering.

This is essential when calls to these functions are located in code paths that are executed at module load time.

This is something that can easily happen when defining models, forms and model forms, because Django implements these such that their fields are actually class-level attributes. For that reason, make sure to use lazy translations in the following cases:

Model fields and relationships `verbose_name` and `help_text` option values

For example, to translate the help text of the *name* field in the following model, do the following:

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

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

You can mark names of **ForeignKey**, **ManyToManyField** or **OneToOneField** relationship as translatable by using their **verbose_name** options:

```
class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name='kinds',
        verbose_name=_('kind'),
    )
```

Just like you would do in **verbose_name** you should provide a lowercase verbose name text for the relation as Django will automatically titlecase it when required.

Model verbose names values

It is recommended to always provide explicit **verbose_name** and **verbose_name_plural** options rather than relying on the fallback English-centric and somewhat naïve determination of verbose names Django performs by looking at the model's class name:

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

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))

    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('my things')
```

Model methods `description` argument to the `@display` decorator

For model methods, you can provide translations to Django and the admin site with the **description** argument to the **display()** decorator:

```
from django.contrib import admin
from django.db import models
from django.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name='kinds',
        verbose_name=_('kind'),
    )

    @admin.display(description=_('Is it a mouse?'))
    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
```

Working with lazy translation objects

The result of a **`gettext_lazy()`** call can be used wherever you would use a string (a **`str`** object) in other Django code, but it may not work with arbitrary Python code. For example, the following won't work because the requests library doesn't handle **`gettext_lazy`** objects:

```
body = gettext_lazy("I \u2764 Django")  # (Unicode :heart:)
requests.post('https://example.com/send', data={'body': body})
```

You can avoid such problems by casting **`gettext_lazy()`** objects to text strings before passing them to non-Django code:

```
requests.post('https://example.com/send', data={'body': str(body)})
```

If you don't like the long **`gettext_lazy`** name, you can alias it as **`_`** (underscore), like so:

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

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Using **`gettext_lazy()`** and **`ngettext_lazy()`** to mark strings in models and utility functions is a common operation. When you're working with these objects elsewhere in your code, you should ensure that you don't accidentally convert them to strings, because they should be converted as late as possible (so that the correct locale is in effect). This necessitates the use of the helper function described next.

Lazy translations and plural

When using lazy translation for a plural string (**`n[p]gettext_lazy`**), you generally don't know the **`number`** argument at the time of the string definition. Therefore, you are authorized to pass a key name instead of an integer as the **`number`** argument. Then **`number`** will be looked up in the dictionary under that key during string interpolation. Here's example:

```
from django import forms
from django.core.exceptions import ValidationError
from django.utils.translation import ngettext_lazy

class MyForm(forms.Form):
    error_message = ngettext_lazy("You only provided %(num)d argument",
        "You only provided %(num)d arguments", 'num')

    def clean(self):
        # ...
        if error:
            raise ValidationError(self.error_message % {'num': number})
```

If the string contains exactly one unnamed placeholder, you can interpolate directly with the **`number`** argument:

```
class MyForm(forms.Form):
    error_message = ngettext_lazy(
        "You provided %d argument",
        "You provided %d arguments",
    )

    def clean(self):
        # ...
        if error:
            raise ValidationError(self.error_message % number)
```

Formatting strings: `format_lazy()`

Python's **`str.format()`** method will not work when either the **`format_string`** or any of the arguments to **`str.format()`** contains lazy translation objects. Instead, you can use **`django.utils.text.format_lazy()`**, which creates a lazy object that runs the **`str.format()`** method only when the result is included in a string. For example:

```
from django.utils.text import format_lazy
from django.utils.translation import gettext_lazy
...
name = gettext_lazy('John Lennon')
instrument = gettext_lazy('guitar')
result = format_lazy('{name}: {instrument}', name=name, instrument=instrument)
```

In this case, the lazy translations in **result** will only be converted to strings when **result** itself is used in a string (usually at template rendering time).

Other uses of lazy in delayed translations

For any other case where you would like to delay the translation, but have to pass the translatable string as argument to another function, you can wrap this function inside a lazy call yourself. For example:

```
from django.utils.functional import lazy
from django.utils.safestring import mark_safe
from django.utils.translation import gettext_lazy as _

mark_safe_lazy = lazy(mark_safe, str)
```

And then later:

```
lazy_string = mark_safe_lazy(_("<p>My <strong>string!</strong></p>"))
```

Localized names of languages

get_language_info()

The **get_language_info()** function provides detailed information about languages:

```
>>> from django.utils.translation import activate, get_language_info
>>> activate('fr')
>>> li = get_language_info('de')
>>> print(li['name'], li['name_local'], li['name_translated'], li['bidi'])
German Deutsch Allemand False
```

The **name**, **name_local**, and **name_translated** attributes of the dictionary contain the name of the language in English, in the language itself, and in your current active language respectively. The **bidi** attribute is True only for bi-directional languages.

The source of the language information is the **django.conf.locale** module. Similar access to this information is available for template code. See below.

Internationalization: in template code

Translations in Django templates uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put **{% load i18n %}** toward the top of your template. As with all template tags, this tag needs to be loaded in all templates which use translations, even those templates that extend from other templates which have already loaded the **i18n** tag.

Warning

Translated strings will not be escaped when rendered in a template. This allows you to include HTML in translations, for example for emphasis, but potentially dangerous characters (e.g. **'**) will also be rendered unchanged.

translate template tag

The **{% translate %}** template tag translates either a constant string (enclosed in single or double quotes) or variable content:

```
<title>{% translate "This is the title." %}</title>
<title>{% translate myvar %}</title>
```

If the **noop** option is present, variable lookup still takes place but the translation is skipped. This is useful when "stubbing out" content that will require translation in the future:

```
<title>{% translate "myvar" noop %}</title>
```

Internally, inline translations use an **gettext()** call.

In case a template var (**myvar** above) is passed to the tag, the tag will first resolve such variable to a string at run-time and then look up that string in the message catalogs.

It's not possible to mix a template variable inside a string within **{% translate %}**. If your translations require strings with variables (placeholders), use **{% blocktranslate %}** instead.

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% translate "This is the title" as the_title %}

<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters:

```
{% translate "starting point" as start %}
{% translate "end point" as end %}
{% translate "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktranslate %}Back to '{{ race }}' homepage{% endblocktranslate %}">{{ race }}</a>
</h1>
<p>
{% for stage in tour_stages %}
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divisibleby:2 %}<br>{% else %}, {% endif %}
{% endfor %}
</p>
```

{% translate %} also supports contextual markers using the **context** keyword:

```
{% translate "May" context "month name" %}
```

Changed in Django 3.1:

The **trans** tag was renamed to **translate**. The **trans** tag is still supported as an alias for backwards compatibility.

`blocktranslate` template tag

Contrarily to the **translate** tag, the **blocktranslate** tag allows you to mark complex sentences consisting of literals and variable content for translation by making use of placeholders:

```
{% blocktranslate %}This string will have {{ value }} inside.{% endblocktranslate %}
```

To translate a template expression – say, accessing object attributes or using template filters – you need to bind the expression to a local variable for use within the translation block. Examples:

```
{% blocktranslate with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktranslate %}

{% blocktranslate with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktranslate %}
```

You can use multiple expressions inside a single **blocktranslate** tag:

```
{% blocktranslate with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktranslate %}
```

Note

The previous more verbose format is still supported: **{% blocktranslate with book|title as book_t and author|title as author_t %}**

Other block tags (for example **{% for %}** or **{% if %}**) are not allowed inside a **blocktranslate** tag.

If resolving one of the block arguments fails, **blocktranslate** will fall back to the default language by deactivating the currently active language temporarily with the **deactivate_all()** function.

This tag also provides for pluralization. To use it:

- Designate and bind a counter value with the name **count**. This value will be the one used to select the right plural form.
- Specify both the singular and plural forms separating them with the **{% plural %}** tag within the **{% blocktranslate %}** and **{% endblocktranslate %}** tags.

An example:

```
{% blocktranslate count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktranslate %}
```

A more complex example:

```
{% blocktranslate with amount=article.price count years=i.length %}
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktranslate %}
```

When you use both the pluralization feature and bind values to local variables in addition to the counter value, keep in mind that the **blocktranslate** construct is internally converted to an **ngettext** call. This means the same notes regarding ngettext variables apply.

Reverse URL lookups cannot be carried out within the **blocktranslate** and should be retrieved (and stored) beforehand:

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktranslate %}
This is a URL: {{ the_url }}
{% endblocktranslate %}
```

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% blocktranslate asvar the_title %}The title is {{ title }}.{% endblocktranslate %}
<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters.

{% blocktranslate %} also supports contextual markers using the **context** keyword:

```
{% blocktranslate with name=user.username context "greeting" %}Hi {{ name }}{% endblocktranslate %}
```

Another feature **{% blocktranslate %}** supports is the **trimmed** option. This option will remove newline characters from the beginning and the end of the content of the **{% blocktranslate %}** tag, replace any whitespace at the beginning and end of a line and merge all lines into one using a space character to separate them. This is quite useful for indenting the content of a **{% blocktranslate %}** tag without having the indentation characters end up in the corresponding entry in the PO file, which makes the translation process easier.

For instance, the following **{% blocktranslate %}** tag:

```
{% blocktranslate trimmed %}
  First sentence.
  Second paragraph.
{% endblocktranslate %}
```

will result in the entry **"First sentence. Second paragraph."** in the PO file, compared to **"\n First sentence.\n Second paragraph.\n"**, if the **trimmed** option had not been specified.

Changed in Django 3.1:

The **blocktrans** tag was renamed to **blocktranslate**. The **blocktrans** tag is still supported as an alias for backwards compatibility.

String literals passed to tags and filters

You can translate string literals passed as arguments to tags and filters by using the familiar **_()** syntax:

```
{% some_tag _("Page not found") value|yesno:_("yes,no") %}
```

In this case, both the tag and the filter will see the translated string, so they don't need to be aware of translations.

Note

In this example, the translation infrastructure will be passed the string **"yes,no"**, not the individual strings **"yes"** and **"no"**. The translated string will need to contain the comma so that the filter parsing code knows how to split up the arguments. For example, a German translator might translate the string **"yes,no"** as **"ja,nein"** (keeping the comma intact).

Comments for translators in templates

Just like with Python code, these notes for translators can be specified using comments, either with the **comment** tag:

```
{% comment %}Translators: View verb{% endcomment %}
{% translate "View" %}

{% comment %}Translators: Short intro blurb{% endcomment %}
<p>{% blocktranslate %}A multiline translatable
literal.{% endblocktranslate %}</p>
```

or with the **{# … #}** one-line comment constructs:

```
{# Translators: Label of a button that triggers search #}
<button type="submit">{% translate "Go" %}</button>

{# Translators: This is a text of the base template #}
{% blocktranslate %}Ambiguous translatable block of text{% endblocktranslate %}
```

Note

Just for completeness, these are the corresponding fragments of the resulting **.po** file:

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base template
# path/to/template/file.html:103
msgid "Ambiguous translatable block of text"
msgstr ""
```

Switching language in templates

If you want to select a language within a template, you can use the **language** template tag:

```
{% load i18n %}

{% get_current_language as LANGUAGE_CODE %}
<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% translate "Welcome to our page" %}</p>

{% language 'en' %}
    {% get_current_language as LANGUAGE_CODE %}
    <!-- Current language: {{ LANGUAGE_CODE }} -->
    <p>{% translate "Welcome to our page" %}</p>
{% endlanguage %}
```

While the first occurrence of “Welcome to our page” uses the current language, the second will always be in English.

Other tags

These tags also require a **{% load i18n %}**.

get_available_languages

{% get_available_languages as LANGUAGES %} returns a list of tuples in which the first element is the language code and the second is the language name (translated into the currently active locale).

---

get_current_language

{% get_current_language as LANGUAGE_CODE %} returns the current user’s preferred language as a string. Example: **en-us**. See How Django discovers language preference.

---

get_current_language_bidi

{% get_current_language_bidi as LANGUAGE_BIDI %} returns the current locale’s direction. If **True**, it’s a right-to-left language, e.g. Hebrew, Arabic. If **False** it’s a left-to-right language, e.g. English, French, German, etc.

---

i18n context processor

If you enable the **django.template.context_processors.i18n** context processor, then each **RequestContext** will have access to **LANGUAGES**, **LANGUAGE_CODE**, and **LANGUAGE_BIDI** as defined above.

---

get_language_info

You can also retrieve information about any of the available languages using provided template tags and filters. To get information about a single language, use the **{% get_language_info %}** tag:

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

You can then access the information:

```
Language code: {{ lang.code }}<br>
Name of language: {{ lang.name_local }}<br>
Name in English: {{ lang.name }}<br>
Bi-directional: {{ lang.bidi }}
Name in the active language: {{ lang.name_translated }}
```

`get_language_info_list`

You can also use the **`{% get_language_info_list %}`** template tag to retrieve information for a list of languages (e.g. active languages as specified in **`LANGUAGES`**). See the section about the set_language redirect view for an example of how to display a language selector using **`{% get_language_info_list %}`**.

In addition to **`LANGUAGES`** style list of tuples, **`{% get_language_info_list %}`** supports lists of language codes. If you do this in your view:

```
context = {'available_languages': ['en', 'es', 'fr']}
return render(request, 'mytemplate.html', context)
```

you can iterate over those languages in the template:

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

Template filters

There are also some filters available for convenience:

- **`{{ LANGUAGE_CODE|language_name }}`** ("German")
- **`{{ LANGUAGE_CODE|language_name_local }}`** ("Deutsch")
- **`{{ LANGUAGE_CODE|language_bidi }}`** (False)
- **`{{ LANGUAGE_CODE|language_name_translated }}`** ("německy", when active language is Czech)

Internationalization: in JavaScript code

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a **`gettext`** implementation.
- JavaScript code doesn't have access to **`.po`** or **`.mo`** files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call **`gettext`**, etc., from within JavaScript.

The main solution to these problems is the following **`JavaScriptCatalog`** view, which generates a JavaScript code library with functions that mimic the **`gettext`** interface, plus an array of translation strings.

The `JavaScriptCatalog` view

***class* `JavaScriptCatalog`**

A view that produces a JavaScript code library with functions that mimic the **`gettext`** interface, plus an array of translation strings.

Attributes

`domain`

Translation domain containing strings to add in the view output. Defaults to **`'djangojs'`**.

`packages`

A list of **`application names`**, among installed applications. Those apps should contain a **`locale`** directory. All those catalogs plus all catalogs found in **`LOCALE_PATHS`** (which are always included) are merged into one catalog. Defaults to **`None`**, which means that all available translations from all **`INSTALLED_APPS`** are provided in the JavaScript output.

Example with default values:

```
from django.views.i18n import JavaScriptCatalog

urlpatterns = [
    path('jsi18n/', JavaScriptCatalog.as_view(), name='javascript-catalog'),
]
```

Example with custom packages:

```
urlpatterns = [
    path('jsi18n/myapp/',
         JavaScriptCatalog.as_view(packages=['your.app.label']),
         name='javascript-catalog'),
]
```

If your root URLconf uses **i18n_patterns()**, **JavaScriptCatalog** must also be wrapped by **i18n_patterns()** for the catalog to be correctly generated.

Example with i18n_patterns():

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path('jsi18n/', JavaScriptCatalog.as_view(), name='javascript-catalog'),
)
```

The precedence of translations is such that the packages appearing later in the **packages** argument have higher precedence than the ones appearing at the beginning. This is important in the case of clashing translations for the same literal.

If you use more than one **JavaScriptCatalog** view on a site and some of them define the same strings, the strings in the catalog that was loaded last take precedence.

Using the JavaScript translation catalog

To use the catalog, pull in the dynamically generated script like this:

```
<script src="{% url 'javascript-catalog' %}"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. When the catalog is loaded, your JavaScript code can use the following methods:

- **gettext**
- **ngettext**
- **interpolate**
- **get_format**
- **gettext_noop**
- **pgettext**
- **npgettext**
- **pluralidx**

gettext

The **gettext** function behaves similarly to the standard **gettext** interface within your Python code:

```
document.write(gettext('this is to be translated'));
```

---

ngettext

The **ngettext** function provides an interface to pluralize words and phrases:

```
const objectCount = 1 // or 0, or 2, or 3, ...
const string = ngettext(
    'literal for the singular case',
    'literal for the plural case',
    objectCount
);
```

---

interpolate

The **interpolate** function supports dynamically populating a format string. The interpolation syntax is borrowed from Python, so the **interpolate** function supports both positional and named interpolation:

- Positional interpolation: **obj** contains a JavaScript Array object whose elements values are then sequentially interpolated in their corresponding **fmt** placeholders in the same order they appear. For example:

```
const formats = ngettext(
  'There is %s object. Remaining: %s',
  'There are %s objects. Remaining: %s',
  11
);
const string = interpolate(formats, [11, 20]);
// string is 'There are 11 objects. Remaining: 20'
```

- Named interpolation: This mode is selected by passing the optional boolean **named** parameter as **true**. **obj** contains a JavaScript object or associative array. For example:

```
const data = {
  count: 10,
  total: 50
};

const formats = ngettext(
    'Total: %(total)s, there is %(count)s object',
    'there are %(count)s of a total of %(total)s objects',
    data.count
);
const string = interpolate(formats, data, true);
```

You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code has to make repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with **ngettext** to produce proper pluralizations).

---

get_format

The **get_format** function has access to the configured i18n formatting settings and can retrieve the format string for a given setting name:

```
document.write(get_format('DATE_FORMAT'));
// 'N j, Y'
```

It has access to the following settings:

- **DATE_FORMAT**
- **DATE_INPUT_FORMATS**
- **DATETIME_FORMAT**
- **DATETIME_INPUT_FORMATS**
- **DECIMAL_SEPARATOR**
- **FIRST_DAY_OF_WEEK**
- **MONTH_DAY_FORMAT**
- **NUMBER_GROUPING**
- **SHORT_DATE_FORMAT**
- **SHORT_DATETIME_FORMAT**
- **THOUSAND_SEPARATOR**
- **TIME_FORMAT**
- **TIME_INPUT_FORMATS**
- **YEAR_MONTH_FORMAT**

This is useful for maintaining formatting consistency with the Python-rendered values.

---

gettext_noop

This emulates the **gettext** function but does nothing, returning whatever is passed to it:

```
document.write(gettext_noop('this will not be translated'));
```

This is useful for stubbing out portions of the code that will need translation in the future.

---

pgettext

The **pgettext** function behaves like the Python variant (**pgettext()**), providing a contextually translated word:

```
document.write(pgettext('month name', 'May'));
```

---

npgettext

The **npgettext** function also behaves like the Python variant (**npgettext()**), providing a **pluralized** contextually translated word:

```
document.write(npgettext('group', 'party', 1));
// party
document.write(npgettext('group', 'party', 2));
// parties
```

---

pluralidx

The **pluralidx** function works in a similar way to the **pluralize** template filter, determining if a given **count** should use a plural form of a word or not:

```
document.write(pluralidx(0));
// true
document.write(pluralidx(1));
// false
document.write(pluralidx(2));
// true
```

In the simplest case, if no custom pluralization is needed, this returns **`false`** for the integer **`1`** and **`true`** for all other numbers.

However, pluralization is not this simple in all languages. If the language does not support pluralization, an empty value is provided.

Additionally, if there are complex rules around pluralization, the catalog view will render a conditional expression. This will evaluate to either a **`true`** (should pluralize) or **`false`** (should **not** pluralize) value.

The `JSONCatalog` view

***class* `JSONCatalog`**

In order to use another client-side library to handle translations, you may want to take advantage of the **`JSONCatalog`** view. It's similar to **`JavaScriptCatalog`** but returns a JSON response.

See the documentation for **`JavaScriptCatalog`** to learn about possible values and use of the **`domain`** and **`packages`** attributes.

The response format is as follows:

```
{
   "catalog": {
       # Translations catalog
   },
   "formats": {
       # Language formats for date, time, etc.
   },
   "plural": "..."  # Expression for plural forms, or null.
}
```

Note on performance

The various JavaScript/JSON i18n views generate the catalog from **`.mo`** files on every request. Since its output is constant, at least for a given version of a site, it's a good candidate for caching.

Server-side caching will reduce CPU load. It's easily implemented with the **`cache_page()`** decorator. To trigger cache invalidation when your translations change, provide a version-dependent key prefix, as shown in the example below, or map the view at a version-dependent URL:

```
from django.views.decorators.cache import cache_page
from django.views.i18n import JavaScriptCatalog

# The value returned by get_version() must change when translations change.
urlpatterns = [
    path('jsi18n/',
         cache_page(86400, key_prefix='js18n-%s' % get_version())(JavaScriptCatalog.as_view()),
         name='javascript-catalog'),
]
```

Client-side caching will save bandwidth and make your site load faster. If you're using ETags (**`ConditionalGetMiddleware`**), you're already covered. Otherwise, you can apply conditional decorators. In the following example, the cache is invalidated whenever you restart your application server:

```
from django.utils import timezone
from django.views.decorators.http import last_modified
from django.views.i18n import JavaScriptCatalog

last_modified_date = timezone.now()

urlpatterns = [
    path('jsi18n/',
         last_modified(lambda req, **kw: last_modified_date)(JavaScriptCatalog.as_view()),
         name='javascript-catalog'),
]
```

You can even pre-generate the JavaScript catalog as part of your deployment procedure and serve it as a static file. This radical technique is implemented in django-statici18n.

Internationalization: in URL patterns

Django provides two mechanisms to internationalize URL patterns:

- Adding the language prefix to the root of the URL patterns to make it possible for **`LocaleMiddleware`** to detect the language to activate from the requested URL.
- Making URL patterns themselves translatable via the **`django.utils.translation.gettext_lazy()`** function.

Warning

Using either one of these features requires that an active language be set for each request; in other words, you need to have **`django.middleware.locale.LocaleMiddleware`** in your **`MIDDLEWARE`** setting.

Language prefix in URL patterns

i18n_patterns(**urls*, *prefix_default_language=True*)

This function can be used in a root URLconf and Django will automatically prepend the current active language code to all URL patterns defined within **i18n_patterns()**.

Setting **prefix_default_language** to **False** removes the prefix from the default language (**LANGUAGE_CODE**). This can be useful when adding translations to existing site so that the current URLs won't change.

Example URL patterns:

```
from django.conf.urls.i18n import i18n_patterns
from django.urls import include, path

from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap

urlpatterns = [
    path('sitemap.xml', sitemap, name='sitemap-xml'),
]

news_patterns = ([
    path('', news_views.index, name='index'),
    path('category/<slug:slug>/', news_views.category, name='category'),
    path('<slug:slug>/', news_views.details, name='detail'),
], 'news')

urlpatterns += i18n_patterns(
    path('about/', about_views.main, name='about'),
    path('news/', include(news_patterns, namespace='news')),
)
```

After defining these URL patterns, Django will automatically add the language prefix to the URL patterns that were added by the **i18n_patterns** function. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate('en')
>>> reverse('sitemap-xml')
'/sitemap.xml'
>>> reverse('news:index')
'/en/news/'

>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug': 'news-slug'})
'/nl/news/news-slug/'
```

With **prefix_default_language=False** and **LANGUAGE_CODE='en'**, the URLs will be:

```
>>> activate('en')
>>> reverse('news:index')
'/news/'

>>> activate('nl')
>>> reverse('news:index')
'/nl/news/'
```

Warning

i18n_patterns() is only allowed in a root URLconf. Using it within an included URLconf will throw an **ImproperlyConfigured** exception.

Warning

Ensure that you don't have non-prefixed URL patterns that might collide with an automatically-added language prefix.

Translating URL patterns

URL patterns can also be marked translatable using the **gettext_lazy()** function. Example:

```
from django.conf.urls.i18n import i18n_patterns
from django.urls import include, path
from django.utils.translation import gettext_lazy as _

from about import views as about_views
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    path('sitemap.xml', sitemap, name='sitemap-xml'),
]

news_patterns = ([
    path('', news_views.index, name='index'),
    path(_('category/<slug:slug>/'), news_views.category, name='category'),
    path('<slug:slug>/', news_views.details, name='detail'),
], 'news')

urlpatterns += i18n_patterns(
    path(_('about/'), about_views.main, name='about'),
    path(_('news/'), include(news_patterns, namespace='news')),
)
```

After you've created the translations, the **reverse()** function will return the URL in the active language. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate('en')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/en/news/category/recent/'

>>> activate('nl')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/nl/nieuws/categorie/recent/'
```

Warning

In most cases, it's best to use translated URLs only within a language code prefixed block of patterns (using **i18n_patterns()**), to avoid the possibility that a carelessly translated URL causes a collision with a non-translated URL pattern.

Reversing in templates

If localized URLs get reversed in templates they always use the current language. To link to a URL in another language use the **language** template tag. It enables the given language in the enclosed template section:

```
{% load i18n %}

{% get_available_languages as languages %}

{% translate "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_name }}</a>
    {% endlanguage %}
{% endfor %}
```

The **language** tag expects the language code as the only argument.

Localization: how to create language files

Once the string literals of an application have been tagged for later translation, the translation themselves need to be written (or obtained). Here's how that works.

Message files

The first step is to create a message file for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a **.po** file extension.

Django comes with a tool, **django-admin makemessages**, that automates the creation and upkeep of these files.

Gettext utilities

The **makemessages** command (and **compilemessages** discussed later) use commands from the GNU gettext toolset: **xgettext**, **msgfmt**, **msgmerge** and **msguniq**.

The minimum version of the **gettext** utilities supported is 0.15.

To create or update a message file, run this command:

```
django-admin makemessages -l de
```

…where **de** is the locale name for the message file you want to create. For example, **pt_BR** for Brazilian Portuguese, **de_AT** for Austrian German or **id** for Indonesian.

The script should be run from one of two places:

- The root directory of your Django project (the one that contains **manage.py**).
- The root directory of one of your Django apps.

The script runs over your project source tree or your application source tree and pulls out all strings marked for translation (see How Django discovers translations and be sure **LOCALE_PATHS** is configured correctly). It creates (or updates) a message file in the directory **locale/LANG/LC_MESSAGES**. In the **de** example, the file will be **locale/de/LC_MESSAGES/django.po**.

When you run **makemessages** from the root directory of your project, the extracted strings will be automatically distributed to the proper message files. That is, a string extracted from a file of an app containing a **locale** directory will go in a message file under that directory. A string extracted from a file of an app without any **locale** directory will either go in a message file under the directory listed first in **LOCALE_PATHS** or will generate an error if **LOCALE_PATHS** is empty.

By default **django-admin makemessages** examines every file that has the **.html**, **.txt** or **.py** file extension. If you want to override that default, use the **--extension** or **-e** option to specify the file extensions to examine:

```
django-admin makemessages -l de -e txt
```

Separate multiple extensions with commas and/or use **-e** or **--extension** multiple times:

```
django-admin makemessages -l de -e html,txt -e xml
```

Warning

When creating message files from JavaScript source code you need to use the special **djangojs** domain, **not -e js**.

Using Jinja2 templates?

makemessages doesn't understand the syntax of Jinja2 templates. To extract strings from a project containing Jinja2 templates, use Message Extracting from Babel instead.

Here's an example **babel.cfg** configuration file:

```
# Extraction from Python source files
[python: **.py]

# Extraction from Jinja2 templates
[jinja2: **.jinja]
extensions = jinja2.ext.with_
```

Make sure you list all extensions you're using! Otherwise Babel won't recognize the tags defined by these extensions and will ignore Jinja2 templates containing them entirely.

Babel provides similar features to **makemessages**, can replace it in general, and doesn't depend on **gettext**. For more information, read its documentation about working with message catalogs.

No gettext?

If you don't have the **gettext** utilities installed, **makemessages** will create empty files. If that's the case, either install the **gettext** utilities or copy the English message file (**locale/en/LC_MESSAGES/django.po**) if available and use it as a starting point, which is an empty translation file.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so **makemessages** works, see gettext on Windows for more information.

Each **.po** file contains a small bit of metadata, such as the translation maintainer's contact information, but the bulk of the file is a list of **messages** – mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text **"Welcome to my site."**, like so:

```
_("Welcome to my site.")
```

…then **django-admin makemessages** will have created a **.po** file containing the following snippet – a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

A quick explanation:

- **msgid** is the translation string, which appears in the source. Don't change it.
- **msgstr** is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes, in the form of a comment line prefixed with **#** and located above the **msgid** line, the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the **msgstr** (or **msgid**) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are directly concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

Mind your charset

Due to the way the **gettext** tools work internally and because we want to allow non-ASCII source strings in Django's core and your applications, you **must** use UTF-8 as the encoding for your PO files (the default when PO files are created). This means that everybody will be using the same encoding, which is important when Django processes the PO files.

Fuzzy entries

`makemessages` sometimes generates translation entries marked as fuzzy, e.g. when translations are inferred from previously translated strings. By default, fuzzy entries are **not** processed by `compilemessages`.

To reexamine all source code and templates for new translation strings and update all message files for **all** languages, run this:

```
django-admin makemessages -a
```

Compiling message files

After you create your message file – and each time you make changes to it – you'll need to compile it into a more efficient form, for use by `gettext`. Do this with the `django-admin compilemessages` utility.

This tool runs over all available `.po` files and creates `.mo` files, which are binary files optimized for use by `gettext`. In the same directory from which you ran `django-admin makemessages`, run `django-admin compilemessages` like this:

```
django-admin compilemessages
```

That's it. Your translations are ready for use.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so `django-admin compilemessages` works see gettext on Windows for more information.

.po files: Encoding and BOM usage.

Django only supports `.po` files encoded in UTF-8 and without any BOM (Byte Order Mark) so if your text editor adds such marks to the beginning of files by default then you will need to reconfigure it.

Troubleshooting: `gettext()` incorrectly detects `python-format` in strings with percent signs

In some cases, such as strings with a percent sign followed by a space and a string conversion type (e.g. `_("10% interest")`), `gettext()` incorrectly flags strings with `python-format`.

If you try to compile message files with incorrectly flagged strings, you'll get an error message like `number of format specifications in 'msgid' and 'msgstr' does not match` or `'msgstr' is not a valid Python format string, unlike 'msgid'`.

To workaround this, you can escape percent signs by adding a second percent sign:

```
from django.utils.translation import gettext as _
output = _("10%% interest")
```

Or you can use `no-python-format` so that all percent signs are treated as literals:

```
# xgettext:no-python-format
output = _("10% interest")
```

Creating message files from JavaScript source code

You create and update the message files the same way as the other Django message files – with the `django-admin makemessages` tool. The only difference is you need to explicitly specify what in gettext parlance is known as a domain in this case the `djangojs` domain, by providing a `-d djangojs` parameter, like this:

```
django-admin makemessages -d djangojs -l de
```

This would create or update the message file for JavaScript for German. After updating message files, run `django-admin compilemessages` the same way as you do with normal Django message files.

`gettext` on Windows

This is only needed for people who either want to extract message IDs or compile message files (`.po`). Translation work itself involves editing existing files of this type, but if you want to create your own message files, or want to test or compile a changed message file, download a precompiled binary installer.

You may also use `gettext` binaries you have obtained elsewhere, so long as the `xgettext --version` command works properly. Do not attempt to use Django translation utilities with a `gettext` package if the command `xgettext --version` entered at a Windows command prompt causes a popup window saying "xgettext.exe has generated errors and will be closed by Windows".

Customizing the `makemessages` command

If you want to pass additional parameters to **xgettext**, you need to create a custom **makemessages** command and override its **xgettext_options** attribute:

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):
    xgettext_options = makemessages.Command.xgettext_options + ['--keyword=mytrans']
```

If you need more flexibility, you could also add a new argument to your custom **makemessages** command:

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):

    def add_arguments(self, parser):
        super().add_arguments(parser)
        parser.add_argument(
            '--extra-keyword',
            dest='xgettext_keywords',
            action='append',
        )

    def handle(self, *args, **options):
        xgettext_keywords = options.pop('xgettext_keywords')
        if xgettext_keywords:
            self.xgettext_options = (
                makemessages.Command.xgettext_options[:] +
                ['--keyword=%s' % kwd for kwd in xgettext_keywords]
            )
        super().handle(*args, **options)
```

Miscellaneous

The `set_language` redirect view

set_language(*request*)

As a convenience, Django comes with a view, **django.views.i18n.set_language()**, that sets a user's language preference and redirects to a given URL or, by default, back to the previous page.

Activate this view by adding the following line to your URLconf:

```
path('i18n/', include('django.conf.urls.i18n')),
```

(Note that this example makes the view available at **/i18n/setlang/**.)

Warning

Make sure that you don't include the above URL within **i18n_patterns()** - it needs to be language-independent itself to work correctly.

The view expects to be called via the **POST** method, with a **language** parameter set in request. If session support is enabled, the view saves the language choice in the user's session. It also saves the language choice in a cookie that is named **django_language** by default. (The name can be changed through the **LANGUAGE_COOKIE_NAME** setting.)

After setting the language choice, Django looks for a **next** parameter in the **POST** or **GET** data. If that is found and Django considers it to be a safe URL (i.e. it doesn't point to a different host and uses a safe scheme), a redirect to that URL will be performed. Otherwise, Django may fall back to redirecting the user to the URL from the **Referer** header or, if it is not set, to **/**, depending on the nature of the request:

- If the request accepts HTML content (based on its **Accept** HTTP header), the fallback will always be performed.
- If the request doesn't accept HTML, the fallback will be performed only if the **next** parameter was set. Otherwise a 204 status code (No Content) will be returned.

Changed in Django 3.1:

In older versions, the distinction for the fallback is based on whether the **X-Requested-With** header is set to the value **XMLHttpRequest**. This is set by the jQuery **ajax()** method.

Here's example HTML template code:

```
{% load i18n %}

<form action="{% url 'set_language' %}" method="post">{% csrf_token %}
    <input name="next" type="hidden" value="{{ redirect_to }}">
    <select name="language">
        {% get_current_language as LANGUAGE_CODE %}
        {% get_available_languages as LANGUAGES %}
        {% get_language_info_list for LANGUAGES as languages %}
        {% for language in languages %}
            <option value="{{ language.code }}"{% if language.code == LANGUAGE_CODE %} selected{% endif %}>
                {{ language.name_local }} ({{ language.code }})
            </option>
        {% endfor %}
    </select>
    <input type="submit" value="Go">
</form>
```

In this example, Django looks up the URL of the page to which the user will be redirected in the **redirect_to** context variable.

Explicitly setting the active language

You may want to set the active language for the current session explicitly. Perhaps a user's language preference is retrieved from another system, for example. You've already been introduced to **django.utils.translation.activate()**. That applies to the current thread only. To persist the language for the entire session in a cookie, set the **LANGUAGE_COOKIE_NAME** cookie on the response:

```
from django.conf import settings
from django.http import HttpResponse
from django.utils import translation
user_language = 'fr'
translation.activate(user_language)
response = HttpResponse(...)
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language)
```

You would typically want to use both: **django.utils.translation.activate()** changes the language for this thread, and setting the cookie makes this preference persist in future requests.

Using translations outside views and templates

While Django provides a rich set of i18n tools for use in views and templates, it does not restrict the usage to Django-specific code. The Django translation mechanisms can be used to translate arbitrary texts to any language that is supported by Django (as long as an appropriate translation catalog exists, of course). You can load a translation catalog, activate it and translate text to language of your choice, but remember to switch back to original language, as activating a translation catalog is done on per-thread basis and such change will affect code running in the same thread.

For example:

```
from django.utils import translation

def welcome_translated(language):
    cur_language = translation.get_language()
    try:
        translation.activate(language)
        text = translation.gettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

Calling this function with the value **'de'** will give you **"Willkommen"**, regardless of **LANGUAGE_CODE** and language set by middleware.

Functions of particular interest are **django.utils.translation.get_language()** which returns the language used in the current thread, **django.utils.translation.activate()** which activates a translation catalog for the current thread, and **django.utils.translation.check_for_language()** which checks if the given language is supported by Django.

To help write more concise code, there is also a context manager **django.utils.translation.override()** that stores the current language on enter and restores it on exit. With it, the above example becomes:

```
from django.utils import translation

def welcome_translated(language):
    with translation.override(language):
        return translation.gettext('welcome')
```

Language cookie

A number of settings can be used to adjust language cookie options:

- **LANGUAGE_COOKIE_NAME**
- **LANGUAGE_COOKIE_AGE**
- **LANGUAGE_COOKIE_DOMAIN**
- **LANGUAGE_COOKIE_HTTPONLY**
- **LANGUAGE_COOKIE_PATH**
- **LANGUAGE_COOKIE_SAMESITE**
- **LANGUAGE_COOKIE_SECURE**

Implementation notes

Specialties of Django translation

Django's translation machinery uses the standard **gettext** module that comes with Python. If you know **gettext**, you might note these specialties in the way Django does translation:

- The string domain is **django** or **djangojs**. This string domain is used to differentiate between different programs that store their data in a common message-file library (usually **/usr/share/locale/**). The **django** domain is used for Python and template translation strings and is loaded into the global translation catalogs. The **djangojs** domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.
- Django doesn't use **xgettext** alone. It uses Python wrappers around **xgettext** and **msgfmt**. This is mostly for convenience.

How Django discovers language preference

Once you've prepared your translations – or, if you want to use the translations that come with Django – you'll need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used – installation-wide, for a particular user, or both.

To set an installation-wide language preference, set **LANGUAGE_CODE**. Django uses this language as the default translation – the final attempt if no better matching translation is found through one of the methods employed by the locale middleware (see below).

If all you want is to run Django with your native language all you need to do is set **LANGUAGE_CODE** and make sure the corresponding message files and their compiled versions (**.mo**) exist.

If you want to let each individual user specify which language they prefer, then you also need to use the **LocaleMiddleware**. **LocaleMiddleware** enables language selection based on data from the request. It customizes content for each user.

To use **LocaleMiddleware**, add **'django.middleware.locale.LocaleMiddleware'** to your **MIDDLEWARE** setting. Because middleware order matters, follow these guidelines:

- Make sure it's one of the first middleware installed.
- It should come after **SessionMiddleware**, because **LocaleMiddleware** makes use of session data. And it should come before **CommonMiddleware** because **CommonMiddleware** needs an activated language in order to resolve the requested URL.
- If you use **CacheMiddleware**, put **LocaleMiddleware** after it.

For example, your **MIDDLEWARE** might look like this:

```
MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
]
```

(For more on middleware, see the middleware documentation.)

LocaleMiddleware tries to determine the user's language preference by following this algorithm:

- First, it looks for the language prefix in the requested URL. This is only performed when you are using the **i18n_patterns** function in your root URLconf. See Internationalization: in URL patterns for more information about the language prefix and how to internationalize URL patterns.
- Failing that, it looks for a cookie.

  The name of the cookie used is set by the **LANGUAGE_COOKIE_NAME** setting. (The default name is **django_language**.)
- Failing that, it looks at the **Accept-Language** HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- Failing that, it uses the global **LANGUAGE_CODE** setting.

Notes:

- In each of these places, the language preference is expected to be in the standard language format, as a string. For example, Brazilian Portuguese is **pt-br**.
- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies **de-at** (Austrian German) but Django only has **de** available, Django uses **de**.
- Only languages listed in the **LANGUAGES** setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set **LANGUAGES** to a list of languages. For example:

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

  This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like **de-ch** or **en-us**).
- If you define a custom **LANGUAGES** setting, as explained in the previous bullet, you can mark the language names as translation strings – but use **gettext_lazy()** instead of **gettext()** to avoid a circular import.

  Here's a sample settings file:

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

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

Once **LocaleMiddleware** determines the user's preference, it makes this preference available as **request.LANGUAGE_CODE** for each **HttpRequest**. Feel free to read this value in your view code. Here's an example:

```
from django.http import HttpResponse

def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note that, with static (middleware-less) translation, the language is in **settings.LANGUAGE_CODE**, while with dynamic (middleware) translation, it's in **request.LANGUAGE_CODE**.

How Django discovers translations

At runtime, Django builds an in-memory unified catalog of literals-translations. To achieve this it looks for translations by following this algorithm regarding the order in which it examines the different file paths to load the compiled message files (**.mo**) and the precedence of multiple translations for the same literal:

1. The directories listed in **LOCALE_PATHS** have the highest precedence, with the ones appearing first having higher precedence than the ones appearing later.
2. Then, it looks for and uses if it exists a **locale** directory in each of the installed apps listed in **INSTALLED_APPS**. The ones appearing first have higher precedence than the ones appearing later.
3. Finally, the Django-provided base translation in **django/conf/locale** is used as a fallback.

See also

The translations for literals included in JavaScript assets are looked up following a similar but not identical algorithm. See **JavaScriptCatalog** for more details.

You can also put custom format files in the `LOCALE_PATHS` directories if you also set `FORMAT_MODULE_PATH`.

In all cases the name of the directory containing the translation is expected to be named using locale name notation. E.g. `de`, `pt_BR`, `es_AR`, etc. Untranslated strings for territorial language variants use the translations of the generic language. For example, untranslated `pt_BR` strings use `pt` translations.

This way, you can write applications that include their own translations, and you can override base translations in your project. Or, you can build a big project out of several apps and put all translations into one big common message file specific to the project you are composing. The choice is yours.

All message file repositories are structured the same way. They are:

- All paths listed in `LOCALE_PATHS` in your settings file are searched for `<language>/LC_MESSAGES/django.(po|mo)`
- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

To create message files, you use the `django-admin makemessages` tool. And you use `django-admin compilemessages` to produce the binary `.mo` files that are used by `gettext`.

You can also run `django-admin compilemessages --settings=path.to.settings` to make the compiler process all the directories in your `LOCALE_PATHS` setting.

Using a non-English base language

Django makes the general assumption that the original strings in a translatable project are written in English. You can choose another language, but you must be aware of certain limitations:

- `gettext` only provides two plural forms for the original messages, so you will also need to provide a translation for the base language to include all plural forms if the plural rules for the base language are different from English.
- When an English variant is activated and English strings are missing, the fallback language will not be the `LANGUAGE_CODE` of the project, but the original strings. For example, an English user visiting a site with `LANGUAGE_CODE` set to Spanish and original strings written in Russian will see Russian text rather than Spanish.

‹ Internationalization and localization

Format localization ›