Chapter 19: Internationalization

Django was originally developed right in the middle of the United States – quite literally, as Lawrence, Kansas, is less than 40 miles from the geographic center of the continental United States. Like most open source projects, though, Django's community grew to include people from all over the globe. As Django's community became increasingly diverse, *internationalization* and *localization* became increasingly important. Because many developers have at best a fuzzy understanding of these terms, we'll define them briefly.

Internationalization refers to the process of designing programs for the potential use of any locale. This includes marking text (such as UI elements and error messages) for future translation, abstracting the display of dates and times so that different local standards may be observed, providing support for differing time zones, and generally making sure that the code contains no assumptions about the location of its users. You'll often see "internationalization" abbreviated *I18N*. (The "18" refers to the number of letters omitted between the initial "I" and the terminal "N.")

Localization refers to the process of actually translating an internationalized program for use in a particular locale. You'll sometimes see "localization" abbreviated as *L10N*.

Django itself is fully internationalized; all strings are marked for translation, and settings control the display of locale-dependent values like dates and times. Django also ships with more than 50 different localization files. If you're not a native English speaker, there's a good chance that Django is already translated into your primary language.

The same internationalization framework used for these localizations is available for you to use in your own code and templates.

To use this framework, you'll need 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."

Django takes care of using these hooks to translate Web applications, on the fly, according to users' language preferences.

Essentially, Django does two things:

- It lets developers and template authors specify which parts of their applications should be translatable.
- It uses that information to translate Web applications for particular users according to their language preferences.

Note

Django's translation machinery uses GNU `gettext` (http://www.gnu.org/software/gettext/) via the standard `gettext` module that comes with Python.

If You Don't Need Internationalization:

Django's internationalization hooks are enabled by default, which incurs a small bit of overhead. If you don't use internationalization, you should set `USE_I18N = False` in your settings file. If `USE_I18N` is set to `False`, then Django will make some optimizations so as not to load the internationalization machinery.

You'll probably also want to remove `'django.core.context_processors.i18n'` from your`TEMPLATE_CONTEXT_PROCESSORS` setting.

The three steps for internationalizing your Django application are:

1. Embed translation strings in your Python code and templates.
2. Get translations for those strings, in whichever languages you want to support.
3. Activate the locale middleware in your Django settings.

We'll cover each one of these steps in detail.

1. How to Specify Translation Strings

Translation strings specify "This text should be translated." These strings can appear in your Python code and templates. It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

In Python Code

Standard Translation

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

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

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


def my_view(request):

    output = _("Welcome to my site.")

    return HttpResponse(output)
```

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

```
from django.utils.translation import ugettext


def my_view(request):
    output = ugettext("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.py makemessages`, won't be able to find these strings. More on `makemessages` later.)

The strings you pass to `_()` or `ugettext()` 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 placeholders (the month and the day) with their positions 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.

Marking Strings as No-Op

Use the function `django.utils.translation.ugettext_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.

Lazy Translation

Use the function `django.utils.translation.ugettext_lazy()` to translate strings lazily – when the value is accessed rather than when the `ugettext_lazy()` function is called.

For example, to translate a model's `help_text`, do the following:

```
from django.utils.translation import ugettext_lazy



class MyThing(models.Model):

    name =
models.CharField(help_text=ugettext_lazy('This is the
help text'))
```

In this example, `ugettext_lazy()` stores 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 template rendering on the Django admin site.

The result of a `ugettext_lazy()` call can be used wherever you would use a unicode string (an object with type `unicode`) in Python. If you try to use it where a bytestring (a `str` object) is expected, things will not work as expected, since a `ugettext_lazy()` object doesn't know how to convert itself to a bytestring. You can't use a unicode string inside a bytestring, either, so this is consistent with normal Python behavior. For example:

```
# This is fine: putting a unicode proxy into a unicode
string.

u"Hello %s" % ugettext_lazy("people")



# This will not work, since you cannot insert a unicode
object

# into a bytestring (nor can you insert our unicode
proxy there)

"Hello %s" % ugettext_lazy("people")
```

If you ever see output that looks like `"hello <django.utils.functional...>"`, you have tried to insert the result of `ugettext_lazy()` into a bytestring. That's a bug in your code.

If you don't like the verbose name `ugettext_lazy`, you can just alias it as _ (underscore), like so:

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


class MyThing(models.Model):

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

Always use lazy translations in Django models. Field names and table names should be marked for translation (otherwise, they won't be translated in the admin interface). This means writing explicit`verbose_name` and `verbose_name_plural` options in the `Meta` class, though, rather than relying on Django's default determination of `verbose_name` and `verbose_name_plural` by looking at the model's class name:

```
from django.utils.translation import ugettext_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 = _('mythings')
```

Pluralization

Use the function `django.utils.translation.ungettext()` to specify pluralized messages. Example:

```
from django.utils.translation import ungettext


def hello_world(request, count):

    page = ungettext('there is %(count)d object',
        'there are %(count)d objects', count) % {

            'count': count,

        }

    return HttpResponse(page)
```

ungettext takes three arguments: the singular translation string, the plural translation string and the number of objects (which is passed to the translation languages as the count variable).

In Template Code

Translation 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.

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

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

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

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

```
{% blocktrans %}This string will have {{ value }}
inside.{% endblocktrans %}
```

To translate a template expression – say, using template filters – you need to bind the expression to a local variable for use within the translation block:

```
{% blocktrans with value|filter as myvar %}

This will have {{ myvar }} inside.

{% endblocktrans %}
```

If you need to bind more than one expression inside a blocktrans tag, separate the pieces with and:

```
{% blocktrans with book|title as book_t and
author|title as author_t %}

This is {{ book_t }} by {{ author_t }}

{% endblocktrans %}
```

To pluralize, specify both the singular and plural forms with the `{% plural %}` tag, which appears within`{% blocktrans %}` and `{% endblocktrans %}`. Example:

```
{% blocktrans count list|length as counter %}

There is only one {{ name }} object.

{% plural %}

There are {{ counter }} {{ name }} objects.

{% endblocktrans %}
```

Internally, all block and inline translations use the appropriate `ugettext` / `ungettext` call.

Each `RequestContext` has access to three translation-specific variables:

- `LANGUAGES` is 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).
- `LANGUAGE_CODE` is the current user's preferred language, as a string. Example: `en-us`. (See "How Django discovers language preference," below.)
- `LANGUAGE_BIDI` is 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.

If you don't use the `RequestContext` extension, you can get those values with three tags:

```
{% get_current_language as LANGUAGE_CODE %}

{% get_available_languages as LANGUAGES %}

{% get_current_language_bidi as LANGUAGE_BIDI %}
```

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

Translation hooks are also available within any template block tag that accepts constant strings. In those cases, just use `_()` syntax to specify a translation string:

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

In this case, both the tag and the filter will see the already-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).

Working With Lazy Translation Objects

Using `ugettext_lazy()` and `ungettext_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 a couple of helper functions.

Joining Strings: string_concat()

Standard Python string joins (`''.join([...])`) will not work on lists containing lazy translation objects. Instead, you can use `django.utils.translation.string_concat()`, which creates a lazy object that concatenates its contents *and* converts them to strings only when the result is included in a string. For example:

```
from django.utils.translation import string_concat

# ...

name = ugettext_lazy(u'John Lennon')

instrument = ugettext_lazy(u'guitar')

result = string_concat([name, ': ', 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).

The allow_lazy() Decorator

Django offers many utility functions (particularly in `django.utils`) that take a string as their first argument and do something to that string. These functions are used by template filters as well as directly in other code.

If you write your own similar functions and deal with translations, you'll face the problem of what to do when the first argument is a lazy translation object. You don't want to convert it to a string immediately, because you might be using this function outside of a view (and hence the current thread's locale setting will not be correct).

For cases like this, use the `django.utils.functional.allow_lazy()` decorator. It modifies the function so that *if* it's called with a lazy translation as the first argument, the function evaluation is delayed until it needs to be converted to a string.

For example:

```
from django.utils.functional import allow_lazy


def fancy_utility_function(s, ...):

    # Do some conversion on string 's'

    # ...

fancy_utility_function =
allow_lazy(fancy_utility_function, unicode)
```

The `allow_lazy()` decorator takes, in addition to the function to decorate, a number of extra arguments (`*args`) specifying the type(s) that the original function can return. Usually, it's enough to include `unicode`here and ensure that your function returns only Unicode strings.

Using this decorator means you can write your function and assume that the input is a proper string, then add support for lazy translation objects at the end.

2. How to Create Language Files

Once you've tagged your strings for later translation, you need to write (or obtain) the language translations themselves. Here's how that works.

Locale restrictions

Django does not support localizing your application into a locale for which Django itself has not been translated. In this case, it will ignore your translation files. If you were to try this and Django supported it, you would inevitably see a mixture of translated strings (from your application) and English strings (from Django itself). If you want to support a locale for your application that is not already part of Django, you'll need to make at least a minimal translation of the Django core.

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.py makemessages`, that automates the creation and upkeep of these files. To create or update a message file, run this command:

```
django-admin.py makemessages -l de
```

...where `de` is the language code for the message file you want to create. The language code, in this case, is in locale format. For example, it's `pt_BR` for Brazilian Portuguese and `de_AT` for Austrian German.

The script should be run from one of three places:

- The root directory of your Django project.
- The root directory of your Django app.
- The root `django` directory (not a Subversion checkout, but the one that is linked-to via `$PYTHONPATH` or is located somewhere on that path). This is only relevant when you are creating a translation for Django itself.

This script runs over your project source tree or your application source tree and pulls out all strings marked for translation. 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`.

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

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

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

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

When creating JavaScript translation catalogs (which we'll cover later in this chapter,) you need to use the special 'djangojs' domain, **not** `-e js`.

No gettext?

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

Working on Windows?

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

The format of `.po` files is straightforward. 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* – simple 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.py 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!

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

```
django-admin.py 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.py 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.py makemessages`, run`django-admin.py compilemessages` like this:

```
django-admin.py compilemessages
```

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

3. How Django Discovers Language Preference

Once you've prepared your translations – or, if you just want to use the translations that come with Django – you'll just 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 other translator finds a translation.

If all you want to do is run Django with your native language, and a language file is available for your language, all you need to do is set `LANGUAGE_CODE`.

If you want to let each individual user specify which language he or she prefers, use `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_CLASSES`setting. Because middleware order matters, you should follow these guidelines:

- Make sure it's one of the first middlewares installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your `MIDDLEWARE_CLASSES` might look like this:

```
MIDDLEWARE_CLASSES = (

'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
)
```

(For more on middleware, see Chapter 17.)

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

- First, it looks for a `django_language` key in the current user's session.
- Failing that, it looks for a cookie.
- 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, it's OK to mark the languages as translation strings – but use a "dummy" `ugettext()` function, not the one in`django.utils.translation`. You should *never* import `django.utils.translation` from within your settings file, because that module in itself depends on the settings, and that would cause a circular import.

  The solution is to use a "dummy" `ugettext()` function. Here's a sample settings file:

```
ugettext = lambda s: s


LANGUAGES = (
    ('de', ugettext('German')),
    ('en', ugettext('English')),
)
```

  With this arrangement, `django-admin.py makemessages` will still find and mark these strings for translation, but the translation won't happen at runtime – so you'll have to remember to wrap the languages in the *real* `ugettext()` in any code that uses `LANGUAGES` at runtime.

- The `LocaleMiddleware` can only select languages for which there is a Django-provided base translation. If you want to provide translations for your application that aren't already in the set of translations in Django's source tree, you'll want to provide at least basic translations for that language. For example, Django uses technical message IDs to translate date formats and time formats – so you will need at least those translations for the system to work correctly.

A good starting point is to copy the English `.po` file and to translate at least the technical messages – maybe the validation messages, too.

Technical message IDs are easily recognized; they're all upper case. You don't translate the message ID as with other messages, you provide the correct local variant on the provided English value. For example, with `DATETIME_FORMAT` (or `DATE_FORMAT` or `TIME_FORMAT`), this would be the format string that you want to use in your language. The format is identical to the format strings used by the `now`template tag.

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 a simple example:

```
def hello_world(request):

    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`.

Using Translations in Your Own Projects

Django looks for translations by following this algorithm:

- First, it looks for a `locale` directory in the application directory of the view that's being called. If it finds a translation for the selected language, the translation will be installed.
- Next, it looks for a `locale` directory in the project directory. If it finds a translation, the translation will be installed.
- Finally, it checks the Django-provided base translation in `django/conf/locale`.

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

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

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`

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

To create message files, you use the same `django-admin.py makemessages` tool as with the Django message files. You only need to be in the right place – in the directory where either the `conf/locale` (in case of the source tree) or the `locale/` (in case of app messages or project messages) directory are located. And you use the same `django-admin.py compilemessages` to produce the binary `django.mo` files that are used by `gettext`.

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

Application message files are a bit complicated to discover – they need the `LocaleMiddleware`. If you don't use the middleware, only the Django message files and project message files will be processed.

Finally, you should give some thought to the structure of your translation files. If your applications need to be delivered to other users and will be used in other projects, you might want to use app-specific translations. But using app-specific translations and project translations could produce weird problems with`makemessages`: `makemessages` will traverse all directories below the current path and so might put message IDs into the project message file that are already in application message files.

The easiest way out is to store applications that are not part of the project (and so carry their own translations) outside the project tree. That way, `django-admin.py makemessages` on the project level will only translate strings that are connected to your explicit project and not strings that are distributed independently.

The `set_language` Redirect View

As a convenience, Django comes with a view, `django.views.i18n.set_language`, that sets a user's language preference and redirects back to the previous page.

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

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

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

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. Otherwise, it saves the language choice in a cookie that is by default named `django_language`. (The name can be changed through the `LANGUAGE_COOKIE_NAME` setting.)

After setting the language choice, Django redirects the user, following this algorithm:

- Django looks for a `next` parameter in the `POST` data.
- If that doesn't exist, or is empty, Django tries the URL in the `Referrer` header.
- If that's empty – say, if a user's browser suppresses that header – then the user will be redirected to`/` (the site root) as a fallback.

Here's example HTML template code:

```
<form action="/i18n/setlang/" method="post">

<input name="next" type="hidden" value="/next/page/" />

<select name="language">

    {% for lang in LANGUAGES %}

    <option value="{{ lang.0 }}">{{ lang.1 }}</option>

    {% endfor %}

</select>

<input type="submit" value="Go" />

</form>
```

Translations and JavaScript

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 `javascript_catalog` View

The main solution to these problems is the `javascript_catalog` view, which sends out a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings. Those translation strings are taken from the application, project or Django core, according to what you specify in either the info_dict or the URL.

You hook it up like this:

```
js_info_dict = {

    'packages': ('your.app.package',),

}



urlpatterns = patterns('',
    (r'^jsi18n/$',
'django.views.i18n.javascript_catalog', js_info_dict),

)
```

Each string in `packages` should be in Python dotted-package syntax (the same format as the strings in`INSTALLED_APPS`) and should refer to a package that contains a `locale` directory. If you specify multiple packages, all those catalogs are merged into one catalog. This is useful if you have JavaScript that uses strings from different applications.

You can make the view dynamic by putting the packages into the URL pattern:

```
urlpatterns = patterns('',

    (r'^jsi18n/(?P<packages>\S+)/$',
'django.views.i18n.javascript_catalog'),

)
```

With this, you specify the packages as a list of package names delimited by '+' signs in the URL. This is especially useful if your pages use code from different apps and this changes often and you don't want to pull in one big catalog file. As a security measure, these values can only be either `django.conf` or any package from the `INSTALLED_APPS` setting.

Using the JavaScript Translation Catalog

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

```
<script type="text/javascript"
src="/path/to/jsi18n/"></script>
```

This is how the admin fetches the translation catalog from the server. When the catalog is loaded, your JavaScript code can use the standard `gettext` interface to access it:

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

There is also an `ngettext` interface:

```
var object_cnt = 1 // or 0, or 2, or 3, ...

s = ngettext('literal for the singular case',

        'literal for the plural case', object_cnt);
```

and even a string interpolation function:

```
function interpolate(fmt, obj, named);
```

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:

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

```
// s 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:

```
d = {
    count: 10
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
'there are %(count)s of a total of %(total)s objects', d.count);
```

```
s = interpolate(fmts, d, 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).

Creating JavaScript Translation Catalogs

You create and update the translation catalogs the same way as the other

Django translation catalogs – with the django-admin.py makemessages tool. The only difference is you need to provide a `-d djangojs` parameter, like this:

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

This would create or update the translation catalog for JavaScript for German. After updating translation catalogs, just run `django-admin.py compilemessages` the same way as you do with normal Django translation catalogs.

Notes for Users Familiar with `gettext`

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.

`gettext` on Windows

This is only needed for people who either want to extract message IDs or compile message files (`.po`). Translation work itself just 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, you will need the `gettext` utilities:

- Download the following zip files from http://sourceforge.net/projects/gettext
- `gettext-runtime-X.bin.woe32.zip`
- `gettext-tools-X.bin.woe32.zip`
- `libiconv-X.bin.woe32.zip`
- Extract the 3 files in the same folder (i.e. `C:\Program Files\gettext-utils`)
- Update the system PATH:
- `Control Panel > System > Advanced > Environment Variables`
- In the `System variables` list, click `Path`, click `Edit`
- Add `;C:\Program Files\gettext-utils\bin` at the end of the `Variable value` field

You may also use `gettext` binaries you have obtained elsewhere, so long as the `xgettext --version`command works properly. Some version 0.14.4 binaries have been found to not support this command. 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".