django-dbgettext Documentation

Release 0.1

Simon Meers

January 04, 2017

Contents

Here lies some basic documentation to get you started with django-dbgettext. The application itself is not overly large or complex, and I recommend perusing the source code for a full understanding.

The premise of django-dbgettext is simple: gettext is already used for translating content from Django's source code and templates, so why not use it for translating database content also? Then the whole process is unified and simplified for translators, and there is no need to provide custom administration interfaces for dynamic content. Simply use the `dbgettext_export` management command to export the content from the database prior to running makemessages.

Contents:

CHAPTER 1

Registering Models

Models can be registered for django-dbgettext in a similar fashion to registering ModelAdmin classes for django.contrib.admin

Simply create a dbgettext_registration.py file within your application root directory, import the dbgettext registry object, and register your Models together with their customised dbgettext.models.Options. For example:

```
from dbgettext.registry import registry, Options
from dbgettext.lexicons import html
from myapp.models import MyModel

class MyModelOptions(Options):
    attributes = ('title',)
    parsed_attributes = {'body': html.lexicon}

registry.register(MyModel, MyModelOptions)
```

That's it. Your dbgettext_registration.py files will be automatically imported by django-dbgettext, and registered models will be included when running dbgettext_export.

You can customise the module name using the DBGETTEXT_REGISTRATION_MODULE_NAME *setting*.

1.1 Project-level Registration

To register models from third-party applications, you can specify a module containing dbgettext registration in the DBGETTEXT_PROJECT_OPTIONS *setting*. For example, the following could be stored in my_project/dbgettext_options.py and DBGETTEXT_PROJECT_OPTIONS set to my_project.dbgettext_options to register flatpages

```
from dbgettext.registry import registry, Options
from dbgettext.lexicons import html
from django.contrib.flatpages.models import FlatPage

class FlatPageOptions(Options):
    attributes = ('title',)
    parsed_attributes = {'content': html.lexicon}

registry.register(FlatPage, FlatPageOptions)
```

1.2 `Options`

- **`attributes:`** tuple of names of fields/callables to be translated
- **`parsed_attributes:`** dictionary of names of fields/callables with HTML content which should have translatable content extracted (should not be listed in `attributes`). Values are callables which take an `Options` argument and return a lexicon suitable for `re.Scanner` – see `dbgettext.lexicons.html` for an example.
- **`translate_if:`** dictionary used to `filter()` queryset
- **`get_path_identifier:`** function returning string used to identify object in path to exported content (given an object)
- **`parent:`**

  name of foreign key to parent model, if registered. Affects:

  - path (path_identifier appended onto parent path)
  - queryset (object only translated if parent is)
- **`custom_lexicon_rules`** list of extra custom rules ((regexp, function) tuples) to be applied when parsing

CHAPTER 2

The `dbgettext_export` Management Command

To obtain a fresh export of your translatable strings from registered models, simply run:

```
python manage.py dbgettext_export
```

from your project's root directory.

This will create a hierarchy of static files (stored by default in `<project_root>/locale/dbgettext`, configurable using the `DBGETTEXT_PATH` and `DBGETTEXT_ROOT` settings) containing the translatable strings. E.g.:

```
locale/dbgettext/myapp/mymodel_1/title.py
locale/dbgettext/myapp/mymodel_1/body.py
locale/dbgettext/myapp/mymodel_2/title.py
locale/dbgettext/myapp/mymodel_2/body.py
```

You can then simply run:

```
python manage.py makemessages (...)
```

as per usual, and these strings will be catalogued for you together with the rest of the translatable strings from your code and templates.

Note: the `<DBGETTEXT_PATH>/<DBGETTEXT_ROOT>` directory is purged each time `dbgettext_export` is run to ensure that old data (e.g. from deleted objects) does not persist in the catalogue.

The paths and names of the static files are intentionally verbose to provide the translator with the context of the string they are translating. You can customise the path using the `get_path_identifier` and `parent` attributes of the `Options` class – see Registering Models.

Settings

The following settings can be overridden in your project's `settings.py`:

- `DBGETTEXT_PATH`: path (absolute or relative to project root) where dbgettext_export should store its output. Defaults to `locale`.
- `DBGETTEXT_ROOT`: name of directory within `DBGETTEXT_PATH` (redundancy to provide protection from auto-purging upon export). Defaults to `dbgettext`.
- `DBGETTEXT_REGISTRATION_MODULE_NAME`: name of modules within apps for registration. Defaults to `dbgettext_registration`
- `DBGETTEXT_PROJECT_OPTIONS`: dotted path to python module containing project-level registration code (e.g. `my_project.dbgettext_options` if you have a `dbgettext_options.py` file in your `my_project` directory).
- `DBGETTEXT_INLINE_HTML_TAGS`: tuple of tag names allowed to appear inline within strings parsed with `dbgettext.lexicons.html`. Defaults to `('b','i','u','em','strong',)`.
- `DBGETTEXT_SPLIT_SENTENCES`: split chunks of text into separate sentences for translation where appropriate. Defaults to `True`.
- `DBGETTEXT_SENTENCE_RE`: compiled regular expression for splitting sentences. Defaults to `re.compile(r'^(.*?\S[\!\?\.])(\s+)(\S+.*)$', re.DOTALL)`.

CHAPTER 4

Parsing Content

4.1 HTML

django-dbgettext comes with HTML parsing functionality out of the box, allowing translatable strings to be extracted from fields with HTML content. To translate an field containing HTML, simply include its name in the `parsed_attributes` dictionary of the registered `Options` (see *Options*), (together with `dbgettext.lexicons.html.lexicon`).

The `DBGETTEXT_INLINE_HTML_TAGS` *setting* can be used to define which HTML tags are allowed to appear within translatable strings. E.g.:

```
This <b>string</b> is <i>translatable</i> by <u>default</u>.
```

The `custom_lexicon_rules` *option* allow the HTML parsing algorithm to be customised to suit your needs. For example, the following `dbgettext_registration.py` file allows images to appear as moveable placeholders in translatable strings:

```
from dbgettext.registry import registry, Options
from dbgettext.parser import Token
from dbgettext.lexicons import html
from models import Text
from django.utils.translation import ugettext as _

class ImageToken(Token):
    """ Allows inline images to be 'translated' as %(image:...)s """

    def __init__(self, raw, src):
        super(ImageToken, self).__init__('image', raw)
        self.src = src

    def is_translatable(self):
        return Token.MAYBE_TRANSLATE

    def get_key(self):
        return 'image:%s' % self.src


class LinkToken(Token):
    """ Allows inline links to be translated as %(link:...)s

    Also demonstrates Token 'inner translation' features using get_raw
    and get_gettext to translate within token itself.
    """

    def __init__(self, raw, href, content):
        super(LinkToken, self).__init__('link', raw)
        self.href = href
        self.content = content

    def is_translatable(self):
        return Token.ALWAYS_TRANSLATE

    def get_raw(self):
        return '<a href="%s">%s</a>' % (_(self.href), _(self.content))

    def get_gettext(self):
        return [self.href, self.content]

    def get_key(self):
        return 'link:%s' % self.content  # should sanitize content first


class TextOptions(Options):
    parsed_attributes = {'body': html.lexicon}

    def image(scanner, token):
        return ImageToken(token, scanner.match.groups()[0])

    def link(scanner, token):
        return LinkToken(token, scanner.match.groups()[0],
                                  scanner.match.groups()[1],)

    custom_lexicon_rules = [
        (r'<img[^>]+src="([^"]+)"[^>]*>', image),
        (r'<a[^>]+href="([^"]+)"[^>]*>([^<]+)</a>', link),
        ]

registry.register(Text, TextOptions)
```

4.2 Subclassing Token

get_key provide this method if your entire `Token` should be be displayed as a placeholder – e.g. `%(get_key_output_here)s`

get_raw provide this method if your `Token` requires inner translation – it should return `self.raw` with any inner translatable parts already gettexted

get_gettext this method should return a list of any translatable strings within your `Token` (again, only required for inner translation)

4.3 Other Parsing?

Not using HTML? Want to parse markdown or something exotic instead? Simply register your own lexicon function like the example provided in `dbgettext.lexicons.html.py` (having read `dbgettext.parser.py` as well).

Once you've got something you're happy with, you may wish to consider submitting your file for inclusion in `dbgettext.lexicons`.

CHAPTER 5

Nested Models

If your application uses models which have parent-child relationships, you may wish to set the `parent` *option* to provide a more appropriate file hierarchy and cascading querysets.

For example, if you have a CMS application with a `Page` model which may include `Link` objects on each page, you could set your `Link Options` like:

```
class LinkOptions(Options):
    parent = 'page'  # name of ForeignKey field to Page
    # other options here...
```

Note that the parent model must also be registered with dbgettext.

This has two benefits:

- child objects will only be translated if their parent is (so, for example, links from an unpublished `Page` will not be included if the parent's `translate-if Option` is set appropriately)
- dbgettext_export will append child output to the parent's path. For example: `locale/dbgettext/cms/page/about_us/contact_us/link_13/` instead of `locale/dbgettext/cms/link/link_13/` – this provides additional context to the translator

Note that the above example uses a customised `get_path_identifier Option` for `Page` to provide nicer a slug-based path (`about_us/contact_us` instead of `page_123`).

CHAPTER 6

Template Tags and Filters

To use dbgettext template tags, simply:

```
{% load dbgettext_tags %}
```

The following template tags/filters are provided

- **`parsed_gettext:`** usage - `object|parsed_gettext:"attribute_name"`

Example Usage: `django.contrib.flatpages`

7.1 Settings

`settings.py`:

```
DBGETTEXT_PROJECT_OPTIONS = 'my_project.dbgettext_options'
```

7.2 Registration

`my_project/dbgettext_options.py`:

```
from dbgettext.registry import registry, Options
from dbgettext.lexicons import html
from django.contrib.flatpages.models import FlatPage

class FlatPageOptions(Options):
    attributes = ('title',)
    parsed_attributes = {'content': html.lexicon}

registry.register(FlatPage, FlatPageOptions)
```

7.3 Template

`templates/flatpages/default.html`:

```
{% load dbgettext_tags i18n %}
<html>
<head>
<title>{% trans flatpage.title %}</title>
</head>
<body>
{{ flatpage|parsed_gettext:"content"|safe }}
</body>
</html>
```

Frequently Asked Questions

Q: dbgettext strings are showing up in my `.po` files, but are not translated when displayed in my browser

A: Make sure you have:

- run `compilemessages`
- restarted the server
- used `gettext` or `trans` to translate the strings in when accessed in your code or templates