i18n

Juan David Ibáñez Palomar*
jdavid@itaapy.com

June 12, 2008

Contents

1 Gettext **2**

2 Message Catalogs. The PO & MO files **2**
2.1 Portable Object . . . 2
2.2 Machine Object . . . 3
2.3 Domains . . . 3

3 The Localization Process **4**
3.1 `igettext-extract.py` . . . 4
3.2 `msgmerge` . . . 5
3.3 The Human Translator . . . 6
3.4 Automatizing the process . . . 6

4 The Source Code. Message Extraction **6**
4.1 Python code . . . 6
4.2 (X)HTML Templates . . . 7

5 The Source Code. Run Time **7**
5.1 Language Negotiation . . . 8
5.2 The templates . . . 9

6 The Build Process **9**

*Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. There is a copy of the license at http://www.gnu.org/copyleft/fdl.html

1 Gettext

The *GNU gettext* utilities[1] provide the basic infrastructure that is used by most Free Software projects to develop and deliver multilingual products; `itools.gettext` is based on this *GNU* standard, and adds some features specifically addressed to the Python developer.

2 Message Catalogs. The PO & MO files

The *GNU gettext* toolset is based on the concept of *message catalogs*. A message catalog keeps text sentences in one language, and their translations to another language. So a product will have as many message catalogs as languages it supports (besides the source language).

The message catalogs are stored in the file system using two related file formats: *PO* (text), and *MO* (binary). The relationship between *PO* and *MO* files is that of a source file to an object file; a *MO* file is generated from a *PO* file.

The *PO* file is the one translators will work with; while the *MO* file is the one that will be used in runtime by the application.

The package `itools.gettext` provides file handlers for both formats.

2.1 Portable Object

The PO acronym stands for *Portable Object*. A PO file looks like this:

```
msgid "Hello World"
msgstr "Hola Mundo"

msgid "The Cock and the Pearl"
msgstr "El Gallo y la Perla"
```

It is basically a collection of sentences in one language and their translation to another language (English to Spanish in the example above).

With the PO handler we can work with the PO file, for example to ask for the translation of a sentence:

```
>>> from itools.gettext import POFile
>>>
>>> po = POFile('es.po')
>>> print po.gettext(u'Hello World')
Hola Mundo
```

The POT file

The POT acronym stands for *Portable Object Template*. The POT file (there is only one per application) is like the PO file, except it has not any translations:

[1] `http://www.gnu.org/software/gettext/`

```
msgid "Hello World"
msgstr ""

msgid "The Cock and the Pearl"
msgstr ""
```

The POT file plays a role in the process to build the PO files from the source code (explained in Section 3).

2.2 Machine Object

The MO acronym stands for *Machine Object*. MO files are binary files generated from PO files. To transform a PO file to a MO file, you can use the command `msgfmt`:

```
$ msgfmt locale/en.po -o locale/en.mo
$ msgfmt locale/es.po -o locale/es.mo
$ msgfmt locale/fr.po -o locale/fr.mo
```

With the MO handler we can get the translation of a sentence from a MO file:

```
>>> from itools.gettext import MOFile
>>>
>>> mo = MOFile('es.mo')
>>> print mo.gettext(u'Hello World')
Hola Mundo
```

2.3 Domains

An application will need to manage one PO file for each language it supports. And for every PO file, there will be a MO file. There will also be one POT file, `locale.pot`. The collection of these files is what we call a domain.

In `itools.gettext` a domain is a folder:

```
$ tree locale
locale
|-- en.mo
|-- en.po
|-- es.mo
|-- es.po
|-- fr.mo
|-- fr.po
`-- locale.pot
```

The class `Domain` allows to us to work with these folders:

```
>>> from itools.gettext import register_domain, get_domain
>>>
>>> register_domain('domain', 'locale')
>>> domain = get_domain('domain')
```

```
>>> print domain.get_languages()
['es', 'en', 'fr']
>>>
>>> print domain.gettext(u'Hello World', 'en')
Hello World
>>> print domain.gettext(u'Hello World', 'fr')
Bonjour le Monde
```

The Registry

As seen in the last example, we have a global registry for domains. This allows, for example, an application to use the domain of `itools`:

```
>>> from itools.gettext import get_domain
>>>
>>> domain = get_domain('itools')
>>> print domain.gettext(u'About', 'fr')
Au sujet de
```

To make your domain globally available, just register it:

```
>>> from itools.gettext import register_domain
>>>
>>> register_domain('my_domain', 'locale')
```

3 The Localization Process

We have seen the core pieces of the puzzle, now we are going to see how they link together. In particular we will study the process by which from the Source Code we produce the translated PO files. Figure 1 shows this process.

First thing to consider is the Source Code. So far we only have made reference to Python modules, but there may be other resources to consider. In a Web Application this probably includes the (X)HTML templates. Section 5 expands on this subject. As far as this Section is concerned, just note that the source code is not only about Python.

3.1 `igettext-extract.py`

There are a number of tools available to extract the text messages from source code and produce the PO files. Most notably the command `xgettext` from the *GNU gettext* toolset; `xgettext` is able to extract messages from many programming languages: C, Python, Lisp, Smalltalk, Java, etc.

However, `itools` includes its own script: `igettext-extract.py`. This script is able to extract messages from some XML languages (unlike `xgettext`), specifically (X)HTML and ODT. Also, `igettext-extract.py` follows a different approach to extract messages from Python code, taking advantage of some characteristics of this language (we will see the details in Section 5).

So here we are to consider the usage of `igettext-extract.py`. But if your needs are different you may use another tool to do the same job.

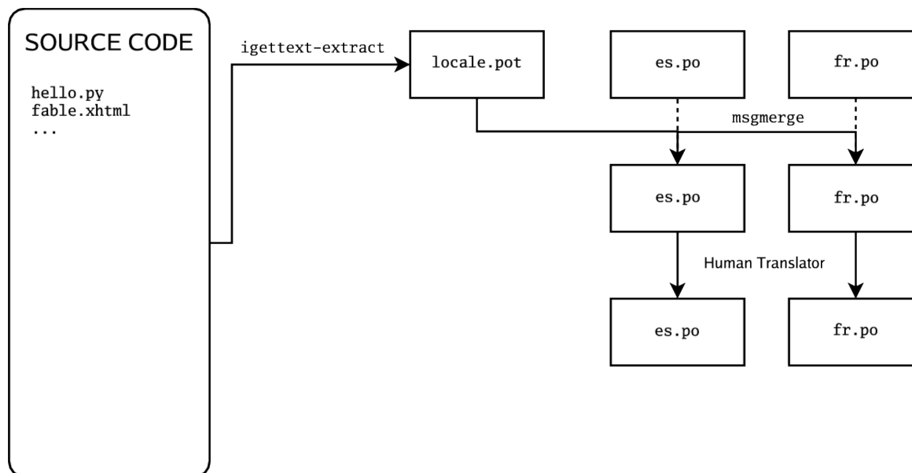

Figure 1: The localization process

From Source Code to Portable Object Template

The first step in the process is to extract the text messages from the Source Code and to produce the `locale.pot` file (as Figure 1 shows). To do so type:

```
$ igettext-extract.py hello.py fable.xhtml.en > locale/locale.pot
```

This command will extract the messages from the source files `fable.xhtml.en` and `hello.py`, and will overwrite the `locale/locale.pot` template file. The `locale.pot` file is automatically generated.

In the same way, you can extract the messages from any ODF [2] document:

```
$ igettext-extract.py document.odt > locale.pot
```

3.2 `msgmerge`

Now we have two scenarios, either we want to add a new translation to our product, or we want to update a translation.

Add a new translation

If we want to add a new translation, it is straightforward, just copy the `locale.pot` file to the new language file. For example:

```
$ cp locale/locale.pot locale/pt.po
```

With this command we have added to our application the translation for the Portuguese language.

[2] Open Document Format

Update a translation

If we want to update an existing translation, it is not harder. We use the `msgmerge` command (from the *GNU gettext* toolset):

```
$ msgmerge -U -s locale/es.po locale/locale.pot
```

This call will preserve the translations already done in the Spanish PO file, and will add the new messages from the `locale.pot` template.

3.3 The Human Translator

Here the work of the *Release Manager* makes a break, and the work of the *Translator* starts.

The translator will edit the PO files and add the missing translations, or correct the inaccurate ones. To do so she will probably use a graphical tool, a very good one is *KBabel*[3].

3.4 Automatizing the process

It is probably a good idea to automatize the chain of calls to `igettext-extract.py` and `msgmerge` on all languages in one shot. To do so one option is to write a `Makefile`.

Another option is to use the `itools` script `isetup-update-locale.py`. But we leave you here to find out how to use it.

4 The Source Code. Message Extraction

For the extraction tool to work properly (e.g. `xgettext` or `igettext-extract.py`), the software developer must follow some rules. These rules depend on the tool being used. Here we are going to explain the rules for `igettext-extract.py`.

We consider our application's user interface is made up of text messages generated from Python code, and of (X)HTML templates. A common configuration in a Web Application written in Python.

4.1 Python code

Unlike `xgettext`, the script `igettext-extract.py` does not require any special markup to find out the Text Strings in the Python code. Because it takes advantage of the fact that Python makes a clear distinction between Text Strings and Byte Strings.

The script `igettext-extract.py` will pick all Unicode string literals it finds in the Python source, but not any byte string literal:

[3] `http://kbabel.kde.org/`

```
# Good
u'Hello World'

# Bad
'Hello World'
```

So the Python developer only needs to properly make the difference between a Text String and a Byte String. Which is always a good idea, even in monolingual applications.

4.2 (X)HTML Templates

With (X)HTML templates `igettext-extract.py` follows a similar approach. This is possible because an (X)HTML file has all the information needed to find out the text messages, without any special markup.

First, there are some attribute values that must be translated, for example the `title` attribute, or the `value` attribute in form buttons. These ones are easy to pick up.

But the big thing are the text nodes. This is a little harder, because there is some markup that makes part of the sentence, like the tags `<em>` and `<strong>`. To solve the problem `igettext-extract.py` uses a simple technique that works rather well. It just makes the distinction between *block-elements*, which delimit a sentence, and *inline-elements*, which belong to the text.

In simple terms: the only rule the developer or the integrator needs to follow is to write correct (X)HTML templates, to use each tag as it is intended to be used.

Moreover, segmentation is applied to the text messages found. This means that a paragraph will be split into its sentences, making the work of the translator much easier.

5 The Source Code. Run Time

Now we need to consider the *run-time* side of the things. This is to say, how we will on run time show the user interface in one language or another.

The first step is to define (register) our application's domain, as we have seen before:

```
from itools.gettext import register_domain

# Register the application's domain
register_domain('my_domain', 'locale')
```

Now we can use the high-level and object-oriented programming interface offered by `itools.gettext`. We define a *domain aware* class by sub-classing from the base class `DomainAware`:

```
class MyApplication(DomainAware):
    class_domain = 'my_domain'

    @classmethod
```

```
    def say_hello(cls):
        print cls.gettext(u'Hello World')
```

Note the way we link the class `MyApplication` to the domain, with the variable `class_domain`.

Now we can use the programming interface provided by the `DomainAware` class. Most notably the class method `gettext`, whose prototype is:

```
gettext(message, language=None, domain=None)
```

The method `gettext` returns the translation of the given *message* to the given *language*, from the given *domain*. By default the domain defined by the `class_domain` variable is used. If the language is not given, the method will try to figure it out, in a process generally known as *language negotiation*.

5.1 Language Negotiation

The default behaviour is to use the system's locale information to find out the users preferred language. For example, the *hello world* example shows this in my notebook:

```
$ python hello.py
Hello World

$ LANG=es python hello.py
Hola Mundo
```

To change the default behaviour we will override the method `select_language`, whose prototype is:

```
select_language(languages)
```

This method must select one language from the list of given languages. For example, if we want to do something really stupid like to choose the language depending on the weekday, we would write:

```
from datetime import date

class MyApplication(DomainAware):
    class_domain = 'my_domain'

    @classmethod
    def select_language(cls, languages):
        weekday = date.today().weekday()
        index = weekday % len(languages)
        return languages[index]
```

If we are developing a Web Application, something more useful would be to look for the user's preferred language in the `Accept-Language` HTTP header. Here the `AcceptLanguage` class from the `itools.i18n` package would be very helpful.

5.2 The templates

The technique we use with `itools` is to have one template per language. We append the language code to the filename:

```
fable.xhtml.en
fable.xhtml.es
fable.xhtml.fr
```

In our example the source template is `fable.xhtml.en`, the others will be generated from this one through the build process. Section 6 explains the details.

So, on *run time* the only thing we need to do is to select the right template:

```
class MyApplication(DomainAware):
    class_domain = 'my_domain'

    @classmethod
    def get_template(cls, name):
        languages = [
            x.rsplit('.', 1)[1] for x in os.listdir('.')
            if x.startswith(name) ]
        language = cls.select_language(languages)
        return '%s.%s' % (name, language)

    @classmethod
    def tell_fable(cls):
        template = cls.get_template('fable.xhtml')
        print open(template).read()
```

In the code above the method `get_template` will figure out the languages available for the desired template, and will choose one by calling `select_language`.

6 The Build Process

To close the chapter we need to address the last point, the process by which we will compile the *Machine Object* files from the *Portable Object* files, and generate the translated templates from the source templates and message catalogs.

To produce the MO files we use the command `msgfmt` from the *GNU gettext* toolset (as we have seen before, in Section 2.2):

```
$ msgfmt locale/es.po -o locale/es.mo
```

To produce the translated templates we use the script `igettext-build.py`:

```
$ igettext-build.py fable.xhtml.en locale/es.po > fable.xhtml.es
```

Here, another example that allow to translate an ODT document. (The translated document will have the same layout than the original one.)

```
$ igettext-build.py document.odt fr.po -o document_fr.odt
```

To automatize the process an option is to write a `Makefile`.