i19

Release 0.1

August 13, 2014

Contents

1 Components 3

2 Workflow 5

3 Features 7

4 Translation Examples 9
- 4.1 Element Content . . . 9
- 4.2 Attributes . . . 9
- 4.3 Nested Translations . . . 10
- 4.4 Pluralization . . . 10
- 4.5 JavaScript . . . 10

5 Installation 13

6 Configuration 15

7 Usage 17

8 Angular Module 19

9 Requirements 21

10 Future Features 23

11 TODO 25

Internationalization tool chain for AngularJS

CHAPTER 1

Components

Angular module:

- Directives that mark translation strings
- Service that provides the translation engine

Translation toolchain:

- String extractor, dumps to gettext POT format (Python)
- gettext PO file to JSON converter (Python)

CHAPTER 2

Workflow

1. Annotate HTML (see Examples below), include `lib/i19.js`
2. Run extraction (see Usage below)
3. Edit translation strings (see `demo/locales/en/LC_MESSAGES/demo.po`)
4. Compile JSON language file (eg `demo/locales/en.json`) and JavaScript pre-loader
5. Use the `$i19` angular service and switch languages on the fly by calling `$i19.set_lang('de')`

CHAPTER 3

Features

Some other features:

- the JSON files with the translation strings can be included in a JS file to be available from the start (think default language) or loaded upon language change (secondary languages)
- app remains single page - no reload upon language change
- full support for angular expressions in i18n strings, eg. `<b>You have {{credits}} EUR left</b>`, with the usual automatic bindings and updates; same for attributes
- error checking upon compilation: introduction of new `{{scope}}` or `<i1i="template">` references
- i18n strings can also contain full html ala `<a ng-click='do()'>foo</a>` (I dont think you would want to use this in general, but experience shows that it might come in handy eventually..)
- supports i18n IDs (named translation strings), reverts to default string as ID
- in case this turns out to be a bad idea way in the future, the syntax is easily converted into Chameleon style i18n attributes for offline processing

All in all the above is a pretty standard gettext workflow. The resulting POs are fully GNU gettext compatible, so you can load them e.g. into Google Translation toolkit and work in a nice UI with automatic translation features and thesaurus and what not. On the other hand, this remains an inline i18n solution, so the HTML inevitably gets more cluttered.

CHAPTER 4

Translation Examples

The following examples demonstrate usage of the various `i19-*` attributes and show the resulting translation data.

4.1 Element Content

To translate the *content* of an HTML element, either use the `i19` tag:

```
<i19>Tag only</i19>
```

i19 ID	Default
Tag only	Tag only

or the `i19` attribute:

```
<div i19>Attribute</div>
```

i19 ID	Default
Attribute	Attribute

To help translators we strongly suggest giving each translation string a name ("i19 ID") to establish context:

```
<div i19="product-view-h">With explicit i18n IDs</div>
```

i19 ID	Default
product-view-h	With explicit i18n IDs

4.2 Attributes

To translate *attributes* of HTML nodes, use the `i19-attr` tag:

```
<img alt="Translated" i19-attr="alt" />
```

i19 ID	Default
Translated	Translated

Again, you should provide a name for the string:

```
<img alt="Translated too" i19-attr="alt portrait-alt" />
```

i19 ID	Default
portrait-alt	Translated too

Multiple attributes can be translated by separating them with commas. You can mix between explicit and implicit i19 IDs:

```
<img alt="Translated" title="Translated too" i19-attr="alt, title with-another-id" />
```

i19 ID	Default
Translated	Translated
with-another-id	Translated too

4.3 Nested Translations

Sometimes you need to translate elements that are contained in other elements that need to be translated as well. Consider this example:

```
<p>Click <a href="..">here</a> to continue</p>
```

While i19 allows you to just include the `<a href..` in a translation string, it is less error-prone if the translator does not have to deal with HTML at all. Ideally, she should translate two strings separately:

- the "here" from the link caption, and
- the surrounding string, preferably with a placeholder: "Click ${placeholder} to continue"

i19 supports this functionality via the `i19-name` attribute:

```
<p i19="outer">Click
    <a i19-name="link-to-next" i19="link-caption">here</a>
    to continue
</p>
```

i19 ID	Default	Translation notes
outer	Click ${link-to-next} to continue	
link-caption	here	Referenced in 'outer' as ${link-to-next}

4.4 Pluralization

i19 has full pluralization support. Just add an Angular Expression as parameter to the i19 ID:

```
<p i19="newmails(count)">You have {{count}} mail</p>
```

i19 ID	Default	Example Translation
newmails(count)[0]	You have {{count}} mail	You have one new Email.
newmails(count)[1]	You have {{count}} mail	You have {{count}} new Emails.

Default pluralization rules are automatically included by PyBabel, and the number of available plurals is adjusted per language in the respective PO file. i19 imports the pluralization function from the PO file.

4.5 JavaScript

Finally, the translation engine can be accessed programmatically from Javascript:

```
alert($i19("Hello World"));
```

Pluralization:

```
alert($i19("You have mail(s)", $scope.count));
```

Hint: If you want Angular-style variable substitution for JavaScript strings, use `$interpolate`:

```
function multi_mail() {
    var translated =
        $i19("You have {{count}} mail", $scope.count);
    return $interpolate(translated)($scope);
}
$scope.mail_counter = 1;
multi_mail() == "You have one Email.";
$scope.mail_counter = 23;
multi_mail() == "You have 23 Emails.";
```

CHAPTER 5

Installation

Install via PIP:

```
pip install https://github.com/johaness/i19/archive/master.zip
```

CHAPTER 6

Configuration

Create a new `Makefile` for your project:

```
# languages to pre-load by including in JavaScript
LANGUAGES_INCLUDE=en

# other languages available as JSON for delayed loading
LANGUAGES_OTHER=de

# translation domain
DOMAIN=my_app

# locale directory, will create one sub-directory per language
LOCALES=locale/

# HTML sources
HTML=*.html

# Output: JavaScript file for pre-loading translation strings
I19JS=locale/i19dict.js

# URL prefix for loading language JSON files
BASEURL=https://cdn.example.org/

include 'i19conf common.mk'
```

Initialize the translation file structure once:

```
make init
```

CHAPTER 7

Usage

Extract strings from source, merge with update existing translations, compile JavaScript and JSON output:

```
make
```

CHAPTER 8

Angular Module

Load the `i19.js` library from the distribution and the `i19dict.js` generated by your Makefile, then access the `$i19` module in angular:

```
angular.module('demo', ['i19']).
    controller('ctl', ['$scope', '$i19', function($scope, $i19) {

        // set language
        $i19.set_lang('de').success(...).error(...);

        // get language
        var l = $i19.get_lang(); // default 'en'

        // enable / disable console warnings for missing translations
        $i19.warn_on_missing_strings = false; // default true

        // return default string in case no translation
        // is available; if false, return i18n ID
        $i19.fallback_default = true;

        // to apply fallback_default, you need generate a language change
        // event:
        $i19.set_lang($i19.get_lang());

    }]);
```

Tip: Run `i19conf i19.js` on the command line to get the full path to the distribution provided `i19.js` file.

CHAPTER 9

Requirements

pybabel, make

CHAPTER 10

Future Features

- Handle multiple occurences of the same translation ID
  - List all filename:lineno
  - Warn if default strings vary
- JS string extractor
  - Check if pybabel parser can be used
- Attribute/Tag name converter for Chameloen to verify fall back
- Manhole with support functions for translators

CHAPTER 11

TODO

- Speed measurements
- Unittests
- Integration tests: HTML source files w/ corner cases
- Documentation