Internationalization

- Internationalization
- How to Internationalize Your Plugin
- Localization
- index

Internationalization

What is Internationalization?

Internationalization is the process of developing a plugin so it can easily be translated into other languages. Internationalization is often abbreviated as `i18n` (because there are 18 letters between the letters i and n).

Why is Internationalization Important?

WordPress is used all over the world, in countries where English is not the main language. The strings in the WordPress plugins need to be coded in a special way so that can be easily translated into other languages. As a developer, you may not be able to provide *localizations* (meaning, changes required to the text and other things like number formats specific to a given *locale* (location)) for all your users; however, a translator can successfully localize the theme without needing to modify the source code itself.

Read further on How to Internationalize your Plugin.

Resources

- Video: i18n: Preparing Your WordPress Theme for the World
- Video: On Internationalization: Plugins and themes for the whole world
- Video: Big in Japan: A Guide for Themes and Internationalization
- Video: Lost in Translation—i18n and WordPress
- Internationalizing And Localizing Your WordPress Theme
- Internationalization: You're probably doing it wrong
- More Internationalization Fun
- Use `wp_localize_script`, It Is Awesome
- Understanding `_n_noop()`

- Language Packs 101 – Prepwork
- Translating WordPress Plugins and Themes: Don't Get Clever
- How to load theme and plugin translations
- Polyglots Handbook: Plugin/Theme Authors Guide

How to Internationalize Your Plugin

In order to make a string translatable in your application you have to wrap the original string in a call to one of a set of special functions. These functions collectively are known as "gettext".

Introduction to Gettext

WordPress uses the gettext libraries and tools for i18n, but not directly: there are a set of special functions created just for the purpose of enabling string translation. These functions are listed below. These are the functions you should use within your plugin.

For a deeper dive into gettext, read the gettext online manual.

Text Domains

Use a *text domain* to denote all text belonging to your plugin. The text domain is a unique identifier to ensure WordPress can distinguish between all loaded translations. This increases portability and plays better with already existing WordPress tools.

The text domain must match the `slug` of the plugin. If your plugin is a single file called `my-plugin.php` or it is contained in a folder called `my-plugin` the domain name must be `my-plugin`. If your plugin is hosted on wordpress.org it must be the slug portion of your plugin URL (`wordpress.org/plugins/my-plugin`).

The text domain name must use dashes and not underscores, be lower case, and have no spaces.

The text domain also needs to be added to the plugin header. WordPress uses it to internationalize your plugin metadata even when the plugin is disabled. The text domain should be same as the one used when loading the text domain.

Header example

```
/*
 * Plugin Name: My Plugin
 * Author: Plugin Author
 * Text Domain: my-plugin
 */
```

[info]Again, change "my-plugin" to the slug of your plugin.[/info]

[info]Since WordPress 4.6 the `Text Domain` header is optional because it must be the same as the plugin slug. There is no harm in including it, but it is not required.[/info]

Domain Path

The domain path defines the location for a plugin's translation. This has a few uses, notably so that WordPress knows where to find the translation even when the plugin is disabled. This defaults to the folder in which your plugin is found.

For example, if the translation is located in a folder called `languages` within your plugin, then the Domain Path is `/languages` and must be written with the first slash:

Header example

```
/*
 * Plugin Name: My Plugin
 * Author: Plugin Author
 * Text Domain: my-plugin
 * Domain Path: /languages
 */
```

[info]The `Domain Path` header can be omitted if the plugin is in the official WordPress Plugin Directory.[/info]

Basic strings

For basic strings (meaning strings without placeholders or plurals) use `__()`. It returns the translation of its argument:

```
__( 'Blog Options', 'my-plugin' );
```

[warning]Do not use variable names or constants for the text domain portion of a gettext function. For example: Do not do this as a shortcut:

`__( 'Translate me.' , $text_domain );`[/warning]

To echo a retrieved tranlsation, use `_e()`. So instead of writing:

```
echo __( 'WordPress is the best!', 'my-plugin' );
```

you can use:

```
_e( 'WordPress is the best!', 'my-plugin' );
```

Variables

What if you have a string like the following:

```
echo 'Your city is $city.'
```

In this case, the `$city` is a variable and should not be part of the translation. The solution is to use placeholders for the variable, along with the `printf` family of functions. Especially helpful are `printf` and `sprintf`. Here is what the right solution looks like:

```
printf(
  /* translators: %s: Name of a city */
  __( 'Your city is %s.', 'my-plugin' ),
  $city
);
```

Notice that here the string for translation is just the template `"Your city is %s."`, which is the same both in the source and at run-time.

Also note that there is a hint for translators so that they know the context of the placeholder.

If you have more than one placeholder in a string, it is recommended that you use argument swapping. In this case, single quotes (`'`) around the string are mandatory because double quotes (`"`) will tell php to interpret the `$s` as the `s` variable, which is not what we want.

```
printf(
  /* translators: 1: Name of a city 2: ZIP code */
  __( 'Your city is %1$s, and your zip code is %2$s.', 'my-plugin' ),
```

```
  $city,
  $zipcode
);
```

Here the zip code is being displayed after the city name. In some languages displaying the zip code and city in opposite order would be more appropriate. Using %s prefix in the above example, allows for such a case. A translation can thereby be written:

```
printf(
  /* translators: 1: Name of a city 2: ZIP code */
  __( 'Your zip code is %2$s, and your city is %1$s.', 'my-plugin' ),
  $city,
  $zipcode
);
```

Important! The following code is incorrect:

```
// This is incorrect do not use.
_e( "Your city is $city.", 'my-plugin' );
```

The strings for translation are extracted from the sources, so the translators will get this phrase to translate: `"Your city is $city."`.

However in the application `_e` will be called with an argument like `"Your city is London."` and `gettext` won't find a suitable translation of this one and will return its argument: `"Your city is London."`. Unfortunately, it isn't translated correctly.

Plurals

Basic Pluralization

If you have string that changes when the number of items changes, you'll need a way to reflect this in your translations. For example, in English you have `"One comment"` and `"Two comments"`. In other languages you can have multiple plural forms. To handle this in WordPress use the `_n()` function.

```
printf(
  _n(
    '%s comment',
    '%s comments',
    get_comments_number(),
```

```
    'my-plugin'
  ),
  number_format_i18n( get_comments_number() )
);
```

`_n()` accepts 4 arguments:

- singular – the singular form of the string (note that it can be used for numbers other than one in some languages, so `'%s item'` should be used instead of `'One item'`).
- plural – the plural form of the string.
- count – the number of objects, which will determine whether the singular or the plural form should be returned (there are languages, which have far more than 2 forms).
- text domain – the plugins text domain.

The return value of the functions is the correct translated form, corresponding to the given count.

Note that some languages use the singular form for other numbers (e.g. 21, 31 and so on, much like '21st', '31st' in English). If you would like to special case the singular, check for it specifically:

```
if ( 1 === $count ) {
  printf( esc_html__( 'Last thing!', 'my-text-domain' ), $count );
} else {
  printf( esc_html( _n( '%d thing.', '%d things.', $count, 'my-text-domain' ) ), $count );
}
```

Also note that the `$count` parameter is often used twice. First `$count` is passed to `_n()` to determine which translated string to use, and then `$count` is passed to `printf()` to substitute the number into the translated string.

Pluralization done later

You first set the plural strings with `_n_noop()` or `_nx_noop()`.

```
$comments_plural = _n_noop(
  '%s comment.',
  '%s comments.'
);
```

Then at a later point in the code you can use `translate_nooped_plural()` to load the strings.

```
printf(
  translate_nooped_plural(
    $comments_plural,
    get_comments_number(),
    'my-plugin'
  ),
  number_format_i18n( get_comments_number() )
);
```

Disambiguation by context

Sometimes one term is used in several contexts and although it is one and the same word in English it has to be translated differently in other languages. For example the word `Post` can be used both as a verb `"Click here to post your comment"` and as a noun `"Edit this post"`. In such cases the `_x()` or `_ex()` function should be used. It is similar to `__()` and `_e()`, but it has an additional argument — the context:

```
_x( 'Post', 'noun', 'my-plugin' );
_x( 'Post', 'verb', 'my-plugin' );
```

Using this method in both cases we will get the string Comment for the original version, but the translators will see two Comment strings for translation, each in the different contexts.

Note that similarly to `__()`, `_x()` has an `echo` version: `_ex()`. The previous example could be written as:

```
_ex( 'Post', 'noun', 'my-plugin' );
_ex( 'Post', 'verb', 'my-plugin' );
```

Use whichever you feel enhances legibility and ease-of-coding.

Descriptions

So that translators know how to translate a string like `__( 'g:i:s a' )` you can add a clarifying comment in the source code. It has to start with the words `translators:` and to be the last PHP comment before the gettext call. Here is an example:

```
/* translators: draft saved date format, see http://php.net/date */
$saved_date_format = __( 'g:i:s a' );
```

It's also used to explain placeholders in a string like `_n_noop( '<strong>Version %1$s</strong> addressed %2$s bug.','<strong>Version %1$s</strong> addressed %2$s bugs.' )`.

```
/* translators: 1: WordPress version number, 2: plural number of bugs. */
_n_noop( '<strong>Version %1$s</strong> addressed %2$s bug.','<strong>Version
%1$s</strong>strong> addressed %2$s bugs.' );
```

Newline characters

Gettext doesn't like `r` (ASCII code: 13) in translatable strings, so please avoid it and use `n` instead.

Empty strings

The empty string is reserved for internal Gettext usage and you must not try to internationalize the empty string. It also doesn't make any sense, because the translators won't see any context.

If you have a valid use-case to internationalize an empty string, add context to both help translators and be in peace with the Gettext system.

Escaping strings

It is good to escape all of your strings, this way the translators cannot run malicious code. There are a few escape functions that are integrated with internationalization functions.

Localization functions

Basic functions

- `__()`
- `_e()`
- `_x()`
- `_ex()`
- `_n()`

- `_nx()`
- `_n_noop()`
- `_nx_noop()`
- `translate_nooped_plural()`

Translate & Escape functions

Strings that require translation and are used in attributes of HTML tags must be escaped.

- `esc_html__()`
- `esc_html_e()`
- `esc_html_x()`
- `esc_attr__()`
- `esc_attr_e()`
- `esc_attr_x()`

Date and number functions

- `number_format_i18n()`
- `date_i18n()`

Best Practices for writing strings

Here are the best practices for writing strings

- Use decent English style – minimize slang and abbreviations.
- Use entire sentences – in most languages word order is different than that in English.
- Split at paragraphs – merge related sentences, but do not include a whole page of text in one string.
- Do not leave leading or trailing whitespace in a translatable phrase.
- Assume strings can double in length when translated.
- Avoid unusual markup and unusual control characters – do not include tags that surround your text.
- Do not put unnecessary HTML markup into the translated string.
- Do not leave URLs for translation, unless they could have a version in another language.
- Add the variables as placeholders to the string as in some languages the placeholders change position.

```
printf(
  __( 'Search results for: %s', 'my-plugin' ),
  get_search_query()
```

```
);
```

- Use format strings instead of string concatenation – translate phrases and not words – `printf( __( 'Your city is %1$s, and your zip code is %2$s.', 'my-plugin' ), $city, $zipcode );` is always better than: `__( 'Your city is ', 'my-plugin' ) . $city . __( ', and your zip code is ', 'my-plugin' ) . $zipcode;`.
- Try to use the same words and same symbols so not multiple strings needs to be translated e.g. `__( 'Posts:', 'my-plugin' );` and `__( 'Posts', 'my-plugin' );`.

Add Text Domain to strings

You must add your Text domain as an argument to every `__()`, `_e()` and `__n()` gettext call, or your translations won't work.

Examples:

- `__( 'Post' )` should become `__( 'Post', 'my-theme' )`.
- `_e( 'Post' )` should become `_e( 'Post', 'my-theme' )`.
- `_n( '%s post', '%s posts', $count )` should become `_n( '%s post', '%s posts', $count, 'my-theme' )`.

If there are strings in your plugin that are also used in WordPress core (e.g. 'Settings'), you should still add your own text domain to them, otherwise they'll become untranslated if the core string changes (which happens).

Adding the text domain by hand can be a burden if not done continuously when writing code and that's why you can do it automatically:

- Download the `add-textdomain.php` script to the folder where the file is you want to add the text domain
- In command line move to the directory where the file is.
- Run this command to create a new file with the text domain added:

```
php add-textdomain.php my-plugin my-plugin.php > new-my-plugin.php
```

If you wish to have the `add-textdomain.php` in a different folder you just need to define the location in the command.

```
php /path/to/add-textdomain.php my-plugin my-plugin.php > new-my-plugin.php
```

Use this command if you don't want a new file outputted:

```
php add-textdomain.php -i my-plugin my-plugin.php
```

If you want to change multiple files in a directory you can also pass a directory to the script:

```
php add-textdomain.php -i my-plugin my-plugin-directory
```

After it's done, the text domain will be added to the end of all gettext calls in the file. If there is an existing text domain it will not be replaced.

Loading Text Domain

Translations can be loaded using `load_plugin_textdomain`, for example:

```
add_action( 'init', 'wpdocs_load_textdomain' );

function wpdocs_load_textdomain() {
  load_plugin_textdomain( 'wpdocs_textdomain', false, dirname( plugin_basename( __FILE__ ) ) .
'/languages' );
}
```

Plugins on WordPress.org

[info]Since WordPress 4.6 translations now take translate.wordpress.org as priority and so plugins that are translated via translate.wordpress.org do not necessary require `load_plugin_textdomain()` anymore. If you don't want to add a `load_plugin_textdomain()` call to your plugin you have to set the `Requires at least:` field in your readme.txt to 4.6 or more.[/info]

If you still want to load your own translations and not the ones from translate, you will have to use a hook filter named `load_textdomain_mofile`.
Example with a .mo file in the `/languages/` directory of your plugin, with this code inserted in the main plugin file:

```
function my_plugin_load_my_own_textdomain( $mofile, $domain ) {
  if ( 'my-domain' === $domain && false !== strpos( $mofile, WP_LANG_DIR . '/plugins/' ) ) {
    $locale = apply_filters( 'plugin_locale', determine_locale(), $domain );
    $mofile = WP_PLUGIN_DIR . '/' . dirname( plugin_basename( __FILE__ ) ) . '/languages/' .
$domain . '-' . $locale . '.mo';
  }
  return $mofile;
```

```
}
add_filter( 'load_textdomain_mofile', 'my_plugin_load_my_own_textdomain', 10, 2 );
```

Handling JavaScript files

Check the Internationalizing javascript section of the Common APIs Handbook to see how to properly load your translation files. There is also the Gutenburg plugin docs page.

Language Packs

If you're interested in language packs and how the import to translate.wordpress.org is working, please read the Meta Handbook page about Translations.

Also refer Plugin/Theme Authors Guide in Polyglots Handbooks for getting your project translated.

Localization

What is localization?

Localization describes the subsequent process of translating an internationalized plugin. Localization is often abbreviated as `l10n` (because there are 10 letters between the l and the n).

Localization files

POT (Portable Object Template) files

This file contains the original strings (in English) in your plugin.

PO (Portable Object) files

Every translator will take the `POT` file and translate the `msgstr` sections into their own language. The result is a `PO` file with the same format as a `POT`, but with translations and some specific headers. There is one `PO` file per language.

MO (Machine Object) files

From every translated `PO` file a `MO` file is built. These are machine-readable, binary files that the gettext functions actually use (they don't care about `.POT` or `.PO` files), and are a "compiled" version of the `PO` file. The conversion is done using the `msgfmt` command line tool. In general, an application may use more than one large logical translatable module and a different `MO` file accordingly. A text domain is a handle of each module, which has a different `MO` file.

Generating the POT file

The `POT` file is the one you need to hand to translators, so that they can do their work. The `POT` and `PO` files can easily be interchangeably renamed to change file types without issues.

[info]It is a good idea to offer the POT file along with your plugin, so that translators won't have to ask you specifically about it.[/info]

There are a couple of ways to generate a `POT` file for your plugin:

WP-CLI

Install WP-CLI and use the `wp i18n make-pot` command according to the documentation.

Poedit

You can also use Poedit locally when translating. This is an open source tool for all major OSs. The free Poedit default version supports manual scanning of all source code with Gettext functions. A pro version of it also features one-click scanning for WordPress plugins. After generating the `PO` file you can rename the file to `POT`. If a `MO` was generated then you can delete that file as it is not needed. If you don't have the pro version you can easily get the Blank POT and use that as the base of your `POT file`. Once you have placed the blank `POT` in the languages folder you can click "Update" in Poedit to update the `POT` file with your strings.

Grunt Tasks

There are even some grunt tasks that you can use to create the POTs. grunt-wp-i18n and grunt-pot. Steps on setting up grunt are beyond the scope of this documentation, but just be aware that it is possible. Here is an example Grunt.js and package.json that you can place in the root of your plugin.

Translate the PO file

Save the translated file as `my-plugin-{locale}.mo`. The locale is the language code and/or country code. For example, the locale for German is `de_DE`. From the code example above the text domain is 'my-plugin' therefore the German MO and PO files should be named `my-plugin-de_DE.mo` and `my-plugin-de_DE.po`. For more information about language and country codes, see Installing WordPress in Your Language.

There are multiple ways to translate a `PO` file.

Manually

You can use a text editor to enter the translation. In a text editor it will look like this.

```
#: plugin-name.php:123
msgid "Page Title"
msgstr ""
```

You enter the translation between the quotation marks. For the German translation it would look like this.

```
#: plugin-name.php:123
msgid "Page Title"
msgstr "Seitentitel"
```

Poedit

You can also use Poedit when translating. The free Poedit default version supports manual scanning of all source code with Gettext functions. A pro version of it also features one-click scanning for WordPress plugins and themes.

Online Services

A third option is to use an online translation service. The general idea is that you upload the `POT` file and then you can give permission to users or translators to translate your plugin. This allows you to track the changes, always have the latest translation and reduce the translation being done twice.

Here are a few tools that can be used to translate PO files online:

- Transifex
- WebTranslateIt
- Poeditor
- GlotPress

Generate MO file

Command line

`msgfmt` is used to create the MO file. `msgfmt` is part of Gettext package. Otherwise command line can be used. A typical `msgfmt` command looks like this:

Unix Operating Systems

```
msgfmt -o filename.mo filename.po
```

Windows Operating Systems

```
msgfmt -o filename.mo filename.po
```

If you have a lot of `PO` files to convert at once, you can run it as a batch. For example, using a `bash` command:

Unix Operating Systems (multiple files)

```
# Find PO files, process each with msgfmt and rename the result to MO
for file in `find . -name "*.po"` ; do msgfmt -o ${file/.po/.mo} $file ; done
```

Windows Operating Systems (multiple files)

For Windows you need to install Cygwin first.

Create a file called `potomo.sh` and put the following into it:

```
#! /bin/sh
# Find PO files, process each with msgfmt and rename the result to MO
for file in `/usr/bin/find . -name '*.po'` ; do /usr/bin/msgfmt -o ${file/.po/.mo} $file ;
done
```

You can run this command in the command line.

```
cd C:/path/to/language/folder/my-plugin/languages & C:/cygwin/bin/bash -c
/cygdrive/c/path/to/script/directory/potomo.sh
```

Poedit

`msgfmt` is also integrated in Poedit allowing you to use it to generate the MO file. There is a setting in the preferences where you can enable or disable it.

Internationalization Localization

Grunt task

There is grunt-po2mo that will convert all of the files.

Tips for Good Translations

Don't translate literally, translate organically

Being multi-lingual, you undoubtedly know that the languages you speak have different structures, rhythms, tones, and inflections. Translated messages don't need to be structured the same way as the English ones: take the ideas that are presented and come up with a message that expresses the same thing in a natural way for the target language. It's the difference between creating an equal message and an equivalent message: don't replicate, replace. Even with more structural items in messages, you have creative license to adapt and change if you feel it will be more logical for, or better adapted to, your target audience.

Try to keep the same level of formality (or informality)

Each message has a different level of formality or informality. Exactly what level of formality or informality to use for each message in your target language is something you'll have to figure out on your own (or with your team), but WordPress messages (informational messages in particular) tend to have a politely informal tone in English. Try to accomplish the equivalent in the target language, within your cultural context.

Don't use slang or audience-specific terms

Some amount of terminology is to be expected in a blog, but refrain from using colloquialisms that only the "in" crowd will get. If the uninitiated blogger were to install WordPress in your language, would they know what the term means? Words like pingback, trackback, and feed are exceptions to this rule; they're terminology that are typically difficult to translate, and many translators choose to leave in English.

Read other software's localizations in your language

If you get stuck or need direction, try reading through the translations of other popular software tools to get a feel for what terms are commonly used, how formality is addressed, etc. Of course, WordPress has its own tone and feel, so keep that in mind when you're reading other localizations, but feel free to dig up UI terms and the like to maintain consistency with other software in your language.

Using Localizations

Place the localization files in the language folder, either in the plugin `languages` folder or as of WordPress 3.7 in the plugin `languages` folder normally under `wp-content`. The full path would be `wp-content/languages/plugins/my-plugin-fr_FR.mo`.

You can change the language in the "General Settings". If you do not see this option, or the language into which you want to switch i snot listed, do it manually:

- Define `WPLANG` inside of `wp-config.php` to your chosen language. For example, if you wanted to use French, you would have: `define ('WPLANG', 'fr_FR');`
- Go to `wp-admin/options-general.php` or "Settings" -> "General"
- Select your language in "Site Language" dropdown
- Go to `wp-admin/update-core.php`
- Click "Update translations", when available
- Core translations files are downloaded, when available

Resources

- Creating .pot file for your theme or plugin
- How To Internationalize WordPress Plugins
- Translating Your Theme
- Blank WordPress POT
- Improved i18n WordPress tools
- How to update translations quickly
- Gist: Complete Localization Grunt task
- WordPress.tv
  - i18n

- internationalization
- translation

index

Internationalization Security

Security is often overlooked when talking about internationalization, but there are a few important things to keep in mind.

Check for Spam and Other Malicious Strings

When a translator submits a localization to you, always check to make sure they didn't include spam or other malicious words in their translation. You can use Google Translate to translate their translation back into your native language so that you can easily compare the original and translated strings.

Escape Internationalized Strings

You can't trust that a translator will only add benign text to their localization; if they want to, they could add malicious JavaScript or other code instead. To protect against that, it's important to treat internationalized strings like you would any other untrusted input.

If you're outputting the strings, then they should be escaped.

Insecure

```
_e( 'The REST API content endpoints were added in WordPress 4.7.', 'your-text-domain' );
```

Secure

```
esc_html_e( 'The REST API content endpoints were added in WordPress 4.7.', 'your-text-domain'
);
```

Alternatively, some people choose to rely on a translation verification mechanism, rather than adding escaping to their code. One example of a verification mechanism is the editor roles that the WordPress Polyglots team uses for translate.wordpress.org. This ensures that any translation submitted by an untrusted contributor has been verified by a trusted editor before being accepted.

Use Placeholders for URLs

Don't include URLs in internationalized strings, because a malicious translator could change them to point to a different URL. Instead, use placeholders for `printf()` or `sprintf()`.

Insecure

```
_e(
  'Please <a href="https://login.wordpress.org/register"> register for a WordPress.org
account</a>.',
  'your-text-domain'
);
```

Secure

```
printf(
  esc_html__( 'Please %1$s register for a WordPress.org account %2$s.', 'your-text-domain' ),
  '<a href="https://login.wordpress.org/register">',
  '</a>'
);
```

Compile Your Own .mo Binaries

Often translators will send the compiled .mo file along with the plaintext .po file, but you should discard their .mo file and compile your own, because you have no way of knowing whether or not it was compiled from the corresponding .po file, or a different one. If it was compiled against a different one, then it could contain spam and other malicious strings without your knowledge.

Using PoEdit to generate the binary will override the headers in the .po file, so instead it's better to compile it from the command line:

```
msgfmt -cv -o /path/to/output.mo /path/to/input.po
```