Languages

Overview

To create your own language file it is necessary that you use the exact contents of the default language file and translate the contents of the define statements. Language files are INI files which are readable by standard text editors and are set up as key/value pairs.

Working With INI Files

INI files have several restrictions. If a value in the ini file contains any non-alphanumeric characters it needs to be enclosed in double-quotes ("). There are also reserved words which must not be used as keys for ini files. These include: NULL, yes, no, TRUE, and FALSE. Values NULL, no and FALSE results in "", yes and TRUE results in 1. Characters {}|&~![()" must not be used anywhere in the key and have a special meaning in the value. Do not use them as it will produce unexpected behavior.

Files are named after their internationally defined standard abbreviation and may include a locale suffix, written as language_REGION. Both the language and region parts are abbreviated to alphabetic, ASCII characters. A user from the USA would expect the language English and the region USA, yielding the locale identifier "en_US". However, a user from the UK may expect a region of UK, yielding "en_UK".

Setup

As previously mentioned, language files are setup as key/value pairs. A key is used within the widget's view and the translator retrieves the associated string for the given language. The following code is an extract from a typical widget language file.

```
; Module - Example (en_US)
MOD_EXAMPLE_HERE_IS_LINE_ONE = "Here is line one"
MOD_EXAMPLE_HERE_IS_LINE_TWO = "Here is line two"
MOD_EXAMPLE_MYLINE = "My Line"
```

Translation keys can be upper or lowercase or a mix of the two and may contain underscores but no spaces. HUBzero convention is to have keys all uppercase with words separated by underscores, following a pattern of {ExtensionPrefix}_{WidgetName}_{TextName} for naming.

Table 1: Translation key prefixes for the various extensionsExtension Type Key Prefix
Component

Extension Type	Key Prefix
Module	
Plugin	
Template	

Adhering to this naming convention is not required but is strongly recommended as it can help avoid potential translation collisions. Since a component can potentially have modules loaded into it, the possibility of a widget and a module having the same translation key arises. To illustrate this, we have the following example of a component named mycomponent that loads a module named mymodule.

The language files for both:

```
; mymodule en_US.ini
MYLINE = "Your Line"
```

```
; mycomponent en_US.ini
MYLINE = "My Line"
```

The layout files for both:

```
<!-- mymodule layout -->
<strong><php echo Lang::txt('MYLINE'); ?></strong>
```

```
<!-- mycomponent layout -->
<div>
 <!-- Load the module -->
 <php echo Module::render('mymodule'); ?>
 <!-- Translate some component text -->
 <php echo Lang::txt('MYLINE'); ?>
</div>
```

Outputs:

```
<div>
 <!-- Load the module -->
 <strong>Your Line</strong>
 <!-- Translate some component text -->
 Your Line
</div>
```

Since the module is loaded in the component view, i.e. *after* the component's translation files have been loaded, the module's instance of MYLINE overwrites the existing MYLINE from the component. Thus, the view outputs "Your Line" for the component translation instead of the expected "My Line". Using the HUBzero naming convention of adding component and module name prefixes helps avoid such errors:

The language files for both:

```
; mymodule en-US.ini
MOD_MYMODULE_MYLINE = "Your Line"
```

```
; mycomponent en-US.ini
COM_MYCOMPONENT_MYLINE = "My Line"
```

The view files for both:

```
<!-- mymodule view -->
<strong><php echo Lang::txt('MOD_MYMODULE_MYLINE'); ?></strong>
```

```
<!-- mycomponent view -->
<div>
 <!-- Load the module -->
 <php echo $this->Widgets()->renderWidget('mywidget'); ?>
 <!-- Translate some module text -->
 <php echo Lang::txt('COM_MYCOMPONENT_MYLINE'); ?>
```

```
</div>
```

Outputs:

```
<div>
 <!-- Load the widget -->
 <strong>Your Line</strong>
 <!-- Translate some module text -->
 My Line
</div>
```

To Further avoid potential collisions as it is possible to have a component and module with the same name, module translation keys are prefixed with MOD_ and component translation keys with COM_.

Translating Text

A translate helper (Lang) is available in all views and the appropriate language file for an extension is preloaded when the extension is instantiated. This is all done automatically and requires no extra work on the developer's part to load and parse translations.

Below is an example of accessing the translate helper:

```
<p><?php echo Lang::txt("MOD_EXAMPLE_MY_LINE"); ?></p>
```

Strings or keys not found in the current translation file will output as is.

Overrides

In order to accommodate rewording across hub deployments, we provide a mechanism for overriding language constants. Web developers are highly encouraged to make use of language constants throughout extension development as language overrides are generally simpler and easier to maintain than view overrides when wording simply needs to be updated.

Administrator Interface

1. On the administrative backend, navigate to the Language Manager through the Extensions menu.

2. Click on the Overrides sub-menu.

3. To add a new Language Override, click the “New” button.

Note: To replace a constant, you can search for a value using the right-hand “*Search Text You Want to Change*” functionality. You will need to provide the wording which you wish to replace. The utility will provide you with the Language Constant to be used in the *Create a New Override* interface

4. Fill out the *Create A New Override* section appropriately.

5. Unless you have another language installed, leave the Language field to its default setting.

- **Note:** The Location field refers to which CMS application this override applies. Possible values are: *Site*, *Administrator*, *Cli*, or *Api*.

- The File field refers to the location of the language override. The format of this would be <webroot>/<hubname>/app/bootstrap/<application>/language/overrides/<language>-override.ini where values for <application> are: *site, administrator, cli,* or *api.*

File Structure

While the administrative backend offers a user-friendly mechanism to add language overrides, system administrators or developers can directly modify the language file.

Language Overrides are application specific. The HUBzero CMS is comprised of four different application types:

- Site - the public-facing interface of the CMS.
- Administrator - the administrative interface of the CMS.
- Cli - The command line interface (muse is a member of this type).
- Api - The REST-ful-esque API for the HUBzero CMS.

When each application is initiated, a set of parameters are “bootstrapped” to the application. These parameters can be defined in the app/bootstrap directory. One of these parameters is

language.

To change a Language Constant on the public-facing Site:

1. Search for the string which you would like to replace.
    1. Using the command `grep` is useful:
        a. Example: grep -i -H -r -n "one more thing" ./
            i. Returns ./core/components/com_projects/site/language/en-GB/en-GB.com_projects.ini:193:COM_PROJECTS_SETUP_BEFORE_COMPLETE="Just one more thing before you get started... (our lawyers made us do it)"
            ii. The constant is: COM_PROJECTS_SETUP_BEFORE_COMPLETE
2. Edit the file: /www/<webroot>/app/bootstrap/site/language/overrides/en-GB.override.ini using your favorite text editor.
3. Insert the constant and language override into this file.
4. Save the file.