Internationalization

If you develop Web applications that have an international target audience, then you have to take internationalization into account. Internationalization is a programming technique that you use to adapt an application for use in an international context. In an internationalized application, important information, such as currency values, numbers, times and dates, specific symbols, native characters, directions, and so on, get rendered in the format and language common to each user's expectations.

There are different ways for implementing multilingual websites in php.

Message Catalog

To incorporate internationalization support into your program, maintain a message catalog of words and phrases and retrieve the appropriate string from the message catalog before printing it. Here's a simple message catalog with foods in American and British English and a function to retrieve words from the catalog:

```
<?php
$messages = array (
   'en_US'=> array(
      'My favorite foods are' =>
         'My favorite foods are',
      'french fries' => 'french fries',
      'biscuit' => 'biscuit',
      'candy' => 'candy',
      'potato chips' => 'potato chips',
      'cookie' => 'cookie',
      'corn' => 'corn',
      'eggplant' => 'eggplant'
   ),

   'en_GB'=> array(
      'My favorite foods are' =>
         'My favourite foods are',
      'french fries' => 'chips',
      'biscuit' => 'scone',
      'candy' => 'sweets',
      'potato chips' => 'crisps',
      'cookie' => 'biscuit',
      'corn' => 'maize',
      'eggplant' => 'aubergine'
   )
);
```

```
function msg($s) {
   global $LANG;
   global $messages;

   if (isset($messages[$LANG][$s])) {
      return $messages[$LANG][$s];
   } else {
      error_log("l10n error:LANG:" .
         "$lang,message:'$s'");
   }
}
?>
```

This short program uses the message catalog to print out a list of foods:

```
<?php
$LANG ='en_GB';

print msg('My favorite foods are').":\n";
print msg('french fries')."\n";
print msg('potato chips')."\n";
print msg('corn')."\n";
print msg('candy')."\n";
?>
```

```
My favourite foods are:
chips
crisps
maize
sweets
```

To have the program output in American English instead of British English, just set $LANG to en_US.

Variable Phrases

You can combine the msg() message retrieval function with printf() to store phrases that require values to be substituted into them. Consider the English sentence "I am 12 years old." In Spanish, the corresponding phrase is "Tengo 12 años." The Spanish phrase can be built by stitching together translations of "I am," the numeral 12, and "years old." It's easier, though, to store them in the message catalogs as printf()-style format strings:

```
<?php
$messages = array(
   'en_US' => array(
```

```
        'I am X years old.' =>
            'I am %d years old.'),
    'es_US' => array(
        'I am X years old.' =>
            'Tengo %d años.')
);
?>
```

You can then pass the results of msg() to printf() as a format string:

```
<?php
$LANG ='es_US';

printf(msg('I am X years old.'), 12);
?>
```

```
Tengo 12 años.
```

For phrases that require the substituted values to be in a different order in different languages, printf() supports changing the order of the arguments:

```
<?php
$messages = array(
    'en_US' => array(
        'I am X years and Y months old.' =>
        'I am %d years and %d months old.'),
    'es_US' => array(
        'I am X years and Y months old.'=>
        'Tengo %2$d meses y %1$d años.')
);
?>
```

With either language, call sprintf() with the same order of arguments (i.e., first years, then months):

```
<?php
$LANG ='es_US';

printf(msg('I am X years and Y months old.'),12,7);
?>
```

```
Tengo 7 meses y 12 años.
```

In the format string, %2$ tells printf() to use the second argument, and %1$ tells it to use the first.

Message Objects

These phrases can also be stored as function return values instead of strings in an array. Storing the phrases as functions removes the need to use printf(). Functions that return a sentence look like this:

```
<?php
// English version
function i_am_X_years_old($age){
   return "I am $age years old.";
}

// Spanish version
function i am_X_years_old($age){
   return "Tengo $age años.";
}
?>
```

If some parts of the message catalog belong in an array, and some parts belong in functions, an object is a helpful container for a language's message catalog. A base object and two simple message catalogs look like this:

```
<?php
class pc_MC_Base {
   var $messages;
   var $lang;

   function msg($s) {
      if (isset($this->messages[$s])) {
         return $this->messages[$s];
      } else {
         error_log("l10n error:LANG:" .
            "$this->lang,message:'$s'");
            }
      }
}

class pc_MC_es_US extends pc_MC_Base {
   function pc_MC_es_US() {
      $this->lang ='es_US';
      $this->messages = array(
         'chicken' => 'pollo',
         'cow' => 'vaca',
         'horse' => 'caballo'
      );
   }
```

```
    function i_am_X_years_old($age){
        return "Tengo $age años";
    }
}

class pc_MC_en_US extends pc_MC_Base {
    function pc_MC_en_US() {
        $this->lang ='en_US';
        $this->messages = array(
            'chicken' => 'chicken',
            'cow' => 'cow',
            'horse' => 'horse'
        );
    }

    function i_am_X_years_old($age) {
        return "I am $age years old.";
    }
}
?>
```

Each message catalog object extends the pc_MC_Base class to get the msg() method, and then defines its own messages (in its constructor) and its own functions that return phrases. Here's how to print text in Spanish:

```
<?php
$MC = new pc_MC_es_US;
print $MC->msg('cow');
print $MC->i_am_X_years_old(15);
?>
```

To print the same text in English, $MC just needs to be instantiated as a pc_MC_en_US object instead of a pc_MC_es_US object. The rest of the code remains unchanged.

Using DB

You can create a separate table in the database for each language. Eg: The English text would be stored in a table called ‘EnglishText’, the German text would be stored in a table called ‘GermanText’ etc.There can be a short key associated with larger text descriptions. A sample code is given below

```
<?   //displays the text for the key welcome_string'
  echo  GetText('welcome_string', $language);
// functions.php file .Function to get text from the database
 function GetText($key,$language)
  {
          // connect to database first.
     //provide the German version of this text
    if ($language == "de")
    {
 $strQuery = "SELECT Description FROM GermanText WHERE Key='$key'";
      $result = mysql_query($strQuery);

      //determine whether this text key exists in the German table
      if (mysql_num_rows($result)!=0)
      {
        $myrow = mysql_fetch_array($result);
        return $myrow["description"];
      }
    }
    //provide the English version of this text
    $strQuery = "SELECT Description FROM EnglishText WHERE Key='$key'";
    $myrow = mysql_fetch_array(mysql_query($strQuery));
    return $myrow["description"];
  }
?>
```

Internationalization Using PHP and GetText

To use gettext in your PHP scripts you'll have to modify all your scripts, however this will have to be done only once. The modifications are not very difficult and you'll probably want to build some utility to scan the code and prompt 'want to modify this?' etc... If you don't want to do that, no problem at all. So what do you have to do in your code? Very simple just replace all the "strings" with something like:

```
print(_("Hello world"));
```

Yes, you use the very unknown PHP function "underscore", which is an alias to the not-short-to-write "gettext" function. Every string that you may translate sometime will have to be translated. Once you get used to this and see the advantages you'll find that you always write your strings wrapped with the "_" function.

Gettext provides a utility called 'xgettext' to extract all the strings from your scripts to a file called a "po" file, you have to use:

```
$xgettext -a src/*.php
```

To extract all the strings. Read the GNU's documentation for 'xgettext' for more options . After xgettexting your code you will have a 'messages.po' file which may look like this one:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR , YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2000-12-08 19:15-0300\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
```

```
"Last-Translator: FULL NAME \n"
"Language-Team: LANGUAGE \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: prueba.php:12
msgid "Hello world"
msgstr ""

#: prueba.php:12 prueba.php:13
msgid "
"
msgstr ""

#: prueba.php:13
msgid "This is a test"
msgstr ""
```

This will be the file that you have to pass to translators, they will have to set msgstr for each entry to the proper translation of the string for the target language. You can add comments (using #comment) for each entry to provide context if needed such as:

```
#: prueba.php:12
# This is displayed at the beginning of the script
msgid "Hello world"
msgstr ""
```

Etc, after this you have a "master" po file where all the msgstr strings are "". Using gettext on a string that is not translated will simply output the msgid. This is a good feature.

Once translators have translated the 'messages.po' file, you'll have several files for different languages. It is time yo use them from PHP.

Producing mo Files

You have to produce a .mo file to use gettext, this is done using the 'msgfmt' command for each .po file:

```
$msgfmt messages.po -o messages.mo
```

Setting up Directories

The best way to use gettext is to build a "locale" directory in some branch of your code tree and inside this directory you create one subdirectory for each language, inside each language's direcory you create an LC_MESSAGES directory, where you put .mo and .po files for the language. Example:

```
/src
/locale/en/LC_MESSAGES/messages.mo
                       messages.po
        es/LC_MESSAGES/messages.po
                       messages.mo
```

Use this URL http://lcweb.loc.gov/standards/iso639-2/bibcodes.html#op to find the 2 character codes for languages. It's nice to follow standards...

You'd probably start liking your "neat" structure of languages a lot, you are ready to decide which language to use in PHP, simply add, in your main script, this code:

```
// Set language to Spanish
putenv ("LC_ALL=es");

// Specify location of translation tables
bindtextdomain ("messages", "./locale");

// Choose domain
textdomain ("messages");
```

The important line is putenv("LC_ALL=es") which basically tells which language to use, with this PHP will use your /locale/es/LC_MESSAGES/messages.mo file to translate all the strings. If you want a different language just change:

```
putenv ("LC_ALL=en);
```

And "auto-magically" you have an English application.

Specifying Character encoding in a html page

A character encoding tells the computer how to interpret raw zeroes and ones into real characters. It usually does this by pairing numbers with characters

Unicode stands for Universal Transformation Format-8 (character encoding)

Unicode based encodings implement Unicode standard and include UTF-8, UTF-16 and UTF-32/UCS-4.

The UTF-8 encoding is variable-width, with each character represented by 1 to 4 bytes. So the first 128 characters (US-ASCII) need one byte. The next 1,920 characters need two bytes to encode. This includes Latin letters with diacritics and characters from Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac and Tāna alphabets. Three bytes are needed for the rest of the Basic Multilingual Plane (which contains virtually all characters in common use). Four bytes are needed for characters in the other planes of Unicode, which include less common CJK characters and various historic scripts.

You can specify the charset in the head section of the html document.

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />