iTranslate4.eu PHP Sample Application

Setup and usage instructions

About the Sample Application

This sample application consist of 2 components.

1. The library that is responsible for the communication with the iTranslate4.eu servers
2. A code sample that demonstrates a simple way to use the library

These are located in the iTranslate and sample folders respectively.

The library

The library communicates with the translation services of iTranslate4.eu providing a simply embeddable solution for anyone who wants to translate phrases, paragraphs or entire documents with PHP.

The code sample

The code sample (in the sample folder) demonstrates how to embed the library into a simple website and explains the steps necessary to configure and use the library. Please consult the index.php file for detailed description.

Prerequisites

The following are prerequisites for the sample application:

- Working PHP 5.3+ installation
- Curl extension for PHP
  - Following configuration option in php.ini

```
extension=php_curl.so
```

- Some knowledge of PHP
- A valid API Key for the iTranslate.eu service

Installation of the PHP Sample

Please follow these step to redy the sample for use:

1. Download the latest version of it from iTranslate4.eu website
2. Extract the archive to any folder to which the webserver has got read access
   Example command on a linux machine:

```
unzip iTranslate4eu-php.zip /home/mywebsite.org/www
```

3. Copy the sample_api_settings.php file to api_settings.php
4. Edit the api_settings.php file and input your API Key and your cache folder so that the configuration file contains the following array definition:

```
$apiSettings = array(
        'apiKey' => 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx',
        'cacheFolder' => 'cache'
    );
```

5. Verify your sample is working by navigating your browser to:

```
http://yourhostname/iTranslateFolder/sample
```

Code Sample Components Explained

The code sample consts of several parts

1. **index.php:** this is PHP code, which handles the initialization of the library, translation requests, and the generation of HTML code to be sent beck to the browser
2. **html_code.php:** this file contains all the HTML markup that is sent to the browser
3. **resources folder:** this folder includes file that are needed for styling the sample code
4. **cache folder:** some of the library function calls are recommended to be cached. These temprary cache files are being saved to this folder.
5. **api_settings.php**: the configuration file that stores your API key and tells the sample code where the cache folder is

Note: How exactly the code sample works is explained in the files themselves in the form of inline comments.

The code sample contains simple functions that should be easy to adopt to any website and thus provides an almost “copy-paste” solution to those who want to use an out of box solution. Even though this works in most situations, it is recommended to read this entire document as it contains pieces of advice regarding security and speed.

Using the Library

If – for some reason – the code sample does not provide you with a flexible enough solution to your needs, the library can be used to create a highly customized application for PHP that communicates with the iTranslate.eu servers.
To get started one only needs to include the required PHP file:

```
require_once '/path/to/ITrsnalteLib/iTranslate.php'
```

The library has got only one configuration option, which is a valid API key. That API Key will be used in all the requests made by the library to the iTranslate servers.

```
// Get an instance of the library - the parameter to this function tells
// where the library is located
$api = iTranslate::getInstance("../");
// set the API key
$api->setApiKey(“xxxxxxxxxxxx”);
```

The library uses a Singleton Pattern meaning that once the API Key has been configured, it is not necessary to set and reset it again. To access iTranslate services anywhere in the code after the API Key has been configured, one can use the following line:

```
$api = iTranslate::getInstance();
```

Translating text

In order to translate some text you need to initialize the library with the method calls written in the previous section. Then you will be able to call the

```
$objTranslation = $api->translate();
```

method to translate the text.

An example to such call could look like this:

```
$objTranslation = $api->translate("en", "de",
    array("I need to be translated to German") );
```

For a complete list of parameters required and optional parameters, take a look at the API Functions section.

API Functions

getLanguages

This function provides information about available source and target languages. These languages are valid options as *src* and *trg* parameters in Translate function.

```
$objAvailableLanguages = $api->getLanguages();
```

Return value:

- Type: iTranslate_Response_Languages
- Defined in file: iTranslate/Response/Languages.php
- Example output:

```
object(iTranslate_Response_Languages)#4 (3) {
  ["src"]=>
  array(1) {
    [0]=>
    string(2) "ar"
  }
  ["trg"]=>
  array(1) {
    [0]=>
    string(2) "en"
  }
  ["exclude"]=>
  array(1) {
    [0]=>
    string(5) "ar,nn"
  }
}
```

getRoutes

This function provides information about the available alternative translations returning the translation route list for a given language pair. This list consists of route IDs which are valid options as *rid* parameter in Translate function.

```
$arrRoutes = $api->getRoutes(“en”, “de”);
```

Return value:

- Type: array
- Example output:

```
Array(1) {
  [0]=>
  string(6) "lin.ts"
}
```

getProviders

This function tells the full name of the different machine translation providers (companies) which take part in the iTranslate4 service.

```
$arrProviders = $api->getProviders();
```

Return value:

- Type: array
- Example output:

```
Array(1) {
  ["lin.ts"]=>
  string(8) "Lingenio"
}
```

translate

The *Translate* function allows the client to send a block of text to the iTranslate4 server for translation using the specified language settings.

```
$objTranslation = $api->translate(“en”, “de”,
    array(“First text to translate.”,
          “Second text block to translate”) );
```

Parameters:

- [REQUIRED] Source Language
- [REQUIRED] Target Language
- [REQUIRED] Text to translate – can be a string or an array of strings
- [OPTIONAL] Minimum translations
  Default: 1
- [OPTIONAL] Maximum translations (Alternative translations)
  Default: 1
- [OPTIONAL] Timeout

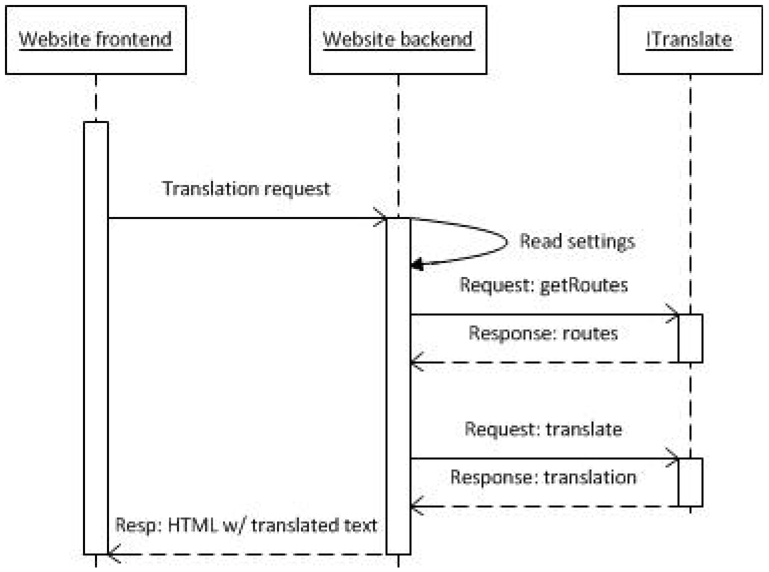
Default: 20 seconds
- [OPTIONAL] Domain of text
  Default: iTranslate_Translate_Domain::DOM_GENERAL
- [OPTIONAL] Routes to take for translation
  Default: null

Return value:
- Type: iTranslate_Response_Translations
- Defined in file: iTranslate/Response/Translations.php
- Example output:

```
object(iTranslate_Response_Translations)#19 (1) {
  ["dat"]=>
  array(2) {
    [0]=>
    object(iTranslate_Response_Translation_Object)#20 (4) {
      ["rid"]=>
      string(6) "lin.ts"
      ["text"]=>
      array(1) {
        [0]=>
        string(68) "Ich möchte diese Probeschnur von Englisch nach 
Deutsch übersetzen."
      }
      ["err"]=>
      NULL
      ["length"]=>
      int(66)
    }
  }
}
```

Communication diagram

The following simplified diagram displays how an average translation request could look like



Security considerations

By placing the library settings file (api_settings.php) into a folder that is publicly readable you might provide the entire world with your api key which should be kept secret.

Speed considerations

Some of the API calls are cachable, not all requests need to be made for every translation. To increase page load times and decrease strain on the server, it is recommended to cache the following functions:

- *iTranslate::getRoutes()*
- *iTranslate::getLanguages()*
- *iTranslate::getProviders()*

The sample application contains a simple file based caching mechanism. It was included in the sample to provide a simple solution to the users, however using APC, Memcached or other caching mechanisms might provide higher performance or conviniance.