Translating your Pas2JS application

Michaël Van Canneyt

October 19, 2022

Abstract

Web Applications and websites are often visible to the world, and allowing the user to use your application or website in his/her own language is clearly improvement of the user interface. Here we show how you can translate your pas2js application.

1 Introduction

If you have a public website or public web application, it is visible to the whole world. Users with any mother tongue will be able to access it, and of course it is more pleasant for them if they can use your site in their familiar language, rather than the lingua franca on the internet: english.

When translating a web application, there are 2 parts to consider when you want the user to be able to use your site:

1. The messages generated from code.
2. The texts that are inserted in the HTML.

Both kinds of translation require a different approach.

2 Translating messages in code

Delphi and Free Pascal have long supported the concept of resource strings: These are named string constants in code, which work almost like real constants. The difference in code is that they are in a 'resourcestring' section in the code instead of a 'const' section. The difference at runtime is that the value of a resource string is fetched at runtime, based on its name. The mechanism to get the value of the string at runtime is pluggable in FPC. You can get the strings from a DLL (the default mechanism in Delphi), or a .po file (common on unix) or invent your own mechanism. By switching the source of the resourcestrings, you can provide a translated version of the strings.

In Pas2JS, the concept of a resourcestring is kept. The generated Javascript contains the default version of the resourcestrings, as defined in the source code. However, value of the strings are kept in a structured manner, and fetched using a special helper function based on the name of the resource string constant. A method is provided to set the values in the javascript sources at runtime: The pas2js RTL comes with a unit that allows you to set the values of the resource strings in the program.

Just as the FPC compiler does, the pas2js compiler can generate a JSON file with the default resource string definitions. There is a command-line option that controls the generation of this file:

-Jrnone no file is generated. This is the default.

-Jrunit The compiler generates a JSON file per used unit.

-Jrprogram The compiler generates a single JSON file with all the used resourcestrings in the program.

The JSON file has extension `jrs`.

The file generated with `-Jrprogram` looks like this:

```
{
  "mystrings" : {
    "Header" : "Translation using resource strings",
    "Paragraph" : "This text will be translated.",
    "TranslateDirect" : "The direct API is used for this example.",
    "TranslateJSON" : "A JSON object is used for this example.",
    "TranslateURL" : "A URL is used for this example.",
    "Button" : "Translate this page"
  },
  "SysUtils" : {
    "SAbortError" : "Operation aborted",
    "SApplicationException" : "Application raised an exception: ",
    "SErrUnknownExceptionType" : "Caught unknown exception type : "
  },
  "translate_basic" : {
    "BasicTitle" : "Translation using resource strings - Basic API"
  }
}
```

As you can see, the strings are grouped in a JSON object per unit, and there is a special section used for the program.

When compiling for a single unit, the following file would be generated for the `mystrings` unit:

```
{
  "mystrings" : {
    "Header" : "Translation using resource strings",
    "Paragraph" : "This text will be translated.",
    "TranslateDirect" : "The direct API is used for this example.",
    "TranslateJSON" : "A JSON object is used for this example.",
    "TranslateURL" : "A URL is used for this example.",
    "Button" : "Translate this page"
  }
}
```

This file can be used to translate the strings in an easy manner using any external tool you want. The JSON format lends itself very well to manipulation using various tools, and merging is easy.

To actually translate the strings, the pas2js RIL contains a unit `rstranslate`, which can be used to translate the strings at runtime.

The unit contains the following definitions:

```
Type

{ TResourceTranslator }
TLoadFailEvent = Reference to Procedure (Sender : TObject;
                                         aCode : Integer;
                                         aError : String);
TOnTranslatedEvent = Reference to Procedure (Sender : TObject;
                                             aURL : String);

TResourceTranslator = Class
  Class Function Instance : TResourceTranslator;
  Procedure Translate(Const aUnit,aString,aTranslation : String);
  Procedure Translate(Const aTranslations : TJSOBject);
  procedure Translate(const aURL: string;
                            aOnTranslated : TOnTranslatedEvent = Nil);
  Procedure ResetTranslation(Const aUnit : String;
                             const aString : String = '');
  \item[OnLoadFail : TLoadFailEvent;
  Property OnURLTranslated : TOnTranslatedEvent;
end;

Function ResourceTranslator : TResourceTranslator;

Procedure Translate(Const aURL : String;
                    aOnTranslated : TOnTranslatedEvent = Nil);
Procedure Translate(Const aUnit,aString,aTranslation : String);
Procedure Translate(Const aTranslations : TJSOBject);

Procedure ResetTranslation(Const aUnit : String;
                           Const aString : String = '');
```

The usage of this unit is quite straightforward:

The `TResourceTranslator` is the class that does the actual work of translating the resourcestring structures generated by the compiler. The unit creates a global instance (available in `TResourceTranslator.Instance`, with the shortcut function `ResourceTranslator`), so you don't need to create an instance.

The `Translate` functions do the actual translation. The first form sets the value of a single resource string:

```
Procedure Translate(Const aUnit,aString,aTranslation : String); overload;
```

The `aUnit` and `aString` parameters identify the resource string, and `aTranslation` is the value to set for the resource string.

The second form accepts a JSON object in the same form as produced by the compiler:

```
Procedure Translate(Const aTranslations : TJSOBject); overload;
```

It will iterate over all strings in the JSON object, and set the translation for every string.

The last form loads a JSON file from a URL, converts the file to a JSON object, and calls the previous function.

```
procedure Translate(const aURL: string; aOnTranslated : TOnTranslatedEvent = Nil); overload;
```

Since loading a file from a URL is asynchronous, there is a callback: when done, the `aOnTranslated` event is called. The objects event `OnURLTranslated` is also called.

If loading of the JSON file fails, the `OnLoadFail` event is called.

The translation mechanism used by Pas2JS does not destroy the original values of the resource strings. They can be reset to their original values if so desired. This can be done with the `ResetTranslation` call. This call can be use to reset the translation for a single unit or for a single resource string.

The translation of the resource strings can of course be done at any moment: as of that moment, using a resource string will from then on use the translated value. This makes the mechanism suitable for the situation where a dropdown is presented to let the the user select his preferred language.

To demonstrate, let's look at the following small webpage, which we will save in a file called `index.html`:

```
<!doctype html>
  <html lang="en">
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="bulma.min.css">
    <title id="translate-title"></title>
    <script src="translate.js"></script>
  </head>

  <body>
    <div class="section">
      <div class="box">
        <h2 class="title is-2" id="translate-header"></h2>
        <p class="block"id="translate-text"></p>
        <button id="btn-translate" class="button is-link"></button>
      </div>
    </div>
    <script>
      window.onload=rtl.run;
    </script>
  </body>
</html>
```

You can see that there are no texts in this HTML page. The texts will be entered in the tags by the program.

Here is part of the program. It has a `mystrings` unit which contains the most of the resource strings. The `PageTitle` resource string is in the program, in order to demonstrate the different sections in which the resource strings are located.

The program has some variables that correspond to the various HTML tags with an ID attribute, the variables are initialized in the `Init` routine:

```
program translation;
{$mode objfpc}
uses
  JS, Classes, SysUtils, Web, mystrings, rstranslate;
```

```
ResourceString
  PageTitle = 'Translation using resource strings - Manual';

Var
  aHeader : TJSHTMLELement;
  aPar : TJSHTMLELement;
  aButton : TJSHTMLButtonELement;
  IsDutch : Boolean;

procedure SetTexts;
begin
  aHeader.InnerHTML:=Header;
  aButton.InnerHTML:=Button;
  aPar.InnerHtml:=Paragraph+' '+TranslateDirect;
  Document.title:=PageTitle;
end;

Procedure Init;

  function GetEl(const aName: string): TJSHTMLELement;
  begin
    Result:=TJSHTMLELement(Document.getElementById(aName));
  end;

begin
  aHeader:=GetEl('translate-header');
  aPar:=GetEl('translate-text');
  aButton:=TJSHTMLButtonELement(GetEl('btn-translate'));
  aButton.onclick:=@DoTranslation;
end;

begin
  Init;
  SetTexts;
end.
```

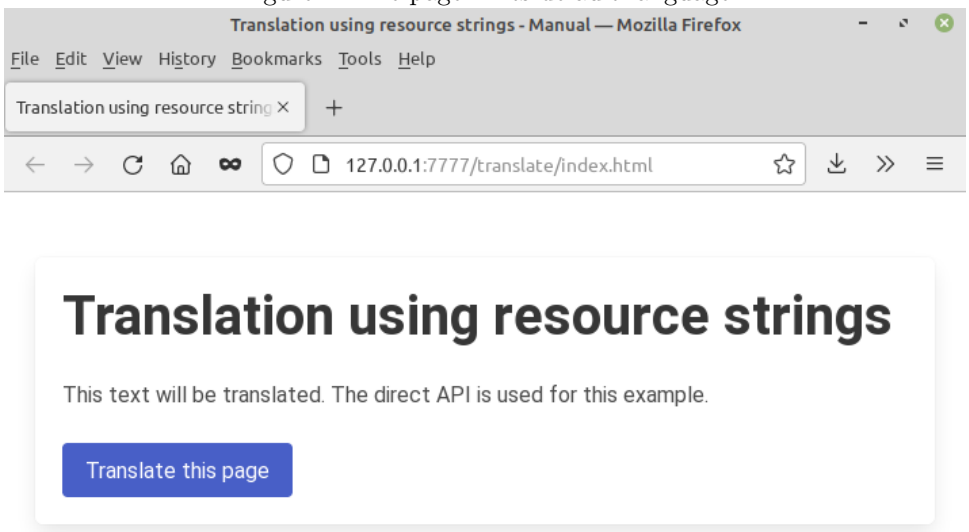
The actual texts of the HTML page are set in the `SetTexts` routine: as you can see, the texts are in constant (resource)strings, located in the `MyStrings` unit. The resulting page looks like figure 1 on page 6.

The `onclick` handler of the `aButton` button (`DoTranslation`) was omitted from the code for clarity. Clicking the button will translate the page from the default (English) to Dutch and vice versa. The handler can be implemented as follows:

```
function DoTranslation(aEvent: TJSMouseEvent): boolean;

begin
  IsDutch:=Not IsDutch;
  if IsDutch then
    begin
    Translate('program','PageTitle','Vertaling met resourcestrings - directe API');
    Translate('mystrings','Button','Vertaal deze pagina');
    Translate('mystrings','Header','Vertaling met resourcestrings');
    Translate('mystrings','Paragraph','Deze tekst wordt vertaald.');
```

Figure 1: The page in its default language




```
    Translate('mystrings','TranslateDirect','De directe API wordt gebruikt voor dit voorbeelo
    Translate('mystrings','TranslateJSON','Een JSON object wordt gebruikt voor dit voorbeeld.
    Translate('mystrings','TranslateURL','Een URL wordt gebruikt voor dit voorbeeld.');
    end
  else
    begin
    // Single string of a module
    ResetTranslation('program','PageTitle');
    // All strings in a module
    ResetTranslation('mystrings');
    end;
  SetTexts;
end;
```

As you can see, all the strings are translated manually to dutch by directly setting the dutch value for the resourcestring using the `translate` call. To go back to English, the `ResetTranslation` routine is called.

At the end, the `SetText` routine is called, which simply re-applies the resource strings. The page will then look like figure 2 on page 7.

Obviously, the above is not a recommended method to translate your program: If you need to support lots of languages and have lots of strings to translate, the amount of code needed to do so would become really big. Rather, it is better to load the translations in some structured format (JSON) and have a small routine that translates the strings. The `rstranslate` routine unit contains a routine which accepts a JSON object with the same format as the file that the compiler creates, and uses that to translate all strings.

One way to do this is to include the translation in the HTML page. For this, we create a small Javascript file that defines the JSON object with the translations:

```
var dutch = {
  "program" : {
```

Figure 2: The page in dutch


```
    "URLTitle":"Vertaling met resourcestrings - URL API"
  },
  "mystrings": {
      "Button":"Vertaal deze pagina",
      "Header":"Vertaling met resourcestrings",
      "Paragraph":"Deze tekst wordt vertaald.",
      "TranslateDirect":"De directe API wordt gebruikt voor dit voorbeeld.",
      "TranslateJSON":"Een JSON object wordt gebruikt voor dit voorbeeld.",
      "TranslateURL":"Een URL wordt gebruikt voor dit voorbeeld."
  }
};
```

We include this in the HTML file by adding the following line:

```
  <script src="dutch.js"></script>
```

and to be able to access the `dutch` variable in our program, we define an external variable in our pascal program as follows:

```
Var
  Dutch : TJSObject; external name 'dutch';
```

The value of the `Dutch` variable is then the JSON object with the translations.

The `DoTranslate` routine now becomes simply the following:

```
function DoTranslation(aEvent: TJSMouseEvent): boolean;

begin
  Result:=True;
  IsDutch:=Not IsDutch;
  if IsDutch then
    Translate(Dutch)
```

```
  else
    begin
    ResetTranslation('program','PageTitle');
    ResetTranslation('mystrings');
    end;
  SetTexts;
end;
```

By defining variables for different languages, one can switch between different languages. Of course, at the startup of the program, all languages will be loaded.

This can be avoided by dynamically injecting the script tag in the HTML as soon as the language needs to be switched: the browser will then load the new translations. By giving all translations the same name for the variable that holds the JSON object, only the last injected language will be kept in memory in the browser.

A disadvantage of this method is still that you need an extra Javascript source file. This necessitates the conversion of a JSON file as produced by the compiler to a Javascript source file. It would be easier if the JSON file could be used directly. Fortunately, this is possible.

The `Translate` call has an overloaded version which can be passed an URL: this URL should point to a JSON file with the translations of all strings, in the format that the compiler provides. The call will download this file and use the translations to translate the strings found in the JSON file. There is callback that can be used to be notified when the translation is complete (downloading a file is asynchronous), this callback can be used to refresh the display.

The third version of our program shows how this can be used. Adapting our program consists of 3 steps:

1. Remove the script tag with the reference to the `dutch.js` file.
2. Remove the external variable definition `Dutch`
3. Change the `DoTranslation` routine to use the URL.

The `DoTranslation` routine will now look as follows:

```
function DoTranslation(aEvent: TJSMouseEvent): boolean;

  Procedure DoTranslated(Sender : TObject; aURL : String);
  begin
    SetTexts;
  end;

begin
  Result:=True;
  IsDutch:=Not IsDutch;
  if IsDutch then
    Translate('dutch.json',@DoTranslated)
  else
    begin
    ResetTranslation('program','PageTitle');
    ResetTranslation('mystrings');
    SetTexts;
    end;
end;
```

Note that the `SetTexts` must be called from the `OnTranslated` callback when loading the JSON file (because only then will the texts actually be translated), but can be called directly when resetting the translations.

3 Translating HTML content

In practice, you will probably not fill up your HTML page with texts from resource strings. One disadvantage is that every location that needs to be translated needs an ID attribute. Second disadvantage is that the Javascript must be loaded before the text appears in your HTML page. If formatting is applied, then the text will be cut into different bits, which is unnatural.

Instead, your HTML will most likely contain the texts in the default language when the page is loaded.

So, how can we translate the texts in the HTML ? There are many possibilities. We could use resource strings and by assigning an ID to all html tags we can translate all texts. This has much of the disadvantages of the previous method. We can make a HTML page for every language; This is a long and cumbersome process.

Here we'll present an alternate approach, implemented in the `Rtl.HTMLTranslate` unit.

The first problem is that we need to know what texts we need to translate. We can simply scan the HTML text and extract all text fragments: this is simple enough.

Then we need to find the translation. Using a hash mechanism, we can quickly search for translations in a file with translations. This has the disadvantage that all context is lost, risking to simply translate wrongly in the case of single words. For example, the english word 'Save' can be translated in several ways to Dutch. Which translation to use depends on the context, and cannot be determined from the hash alone.

A better approach is to give a unique identifier to all texts that needs to be used, so the context of the word or sentence can be looked up. But preferably not the ID, to avoid clashes in case various HTML fragments are combined into a single page. We can use the data attribute for this, it is treated specially in the Javascript DOM API.

Practically, this means that all tags that need translation must be marked as follows:

```
<h2 data-translate="Header">
Translation using HTMLTranslator
</h2>
```

The `data-translate` attribute is set to header `Header`. The translation routine will look for a text named `Header` and when found, will replace the elements `InnerText` or `InnerHTML` with the translated text.

Sometimes, one needs to translate not the inner text, but an attribute of the HTML tag. A prime example is the `placeholder` attribute of the `Input` and `TextArea` tags.

For this, the `Rtl.HTMLTranslate` unit uses a convention: the name of attribute is part of the translate name: everything after the first dash is considered an attribute name.

```
<textarea data-translate="memo-placeholder"
          placeholder="Type your remark here">
```

```
  Translation using HTMLTranslator
</textarea>
```

When the translation engine encounters this, it will look for a translation named 'memo-placeholder' in the translated texts, and will place whatever it finds in the `placeholder` attribute. This convention of course means that you cannot use dashes in your translation names.

The `TextArea` can potentially have 2 texts to translate: the placeholder attribute and the actual text in the text area. Depending on how HTML is used, more attributes may need translation. Since we have only a single data attribute, how can we cater for this ? Simple: the `data-translate` mechanism can handle multiple names, separated by a semicolon.

```
<textarea data-translate="memo;memo-placeholder"
          placeholder="Type your remark here">
  Translation using HTMLTranslator
</textarea>
```

When handling this tag, the translation engine will replace the inner text with the contents of the named translation `memo`, and will replace the `placeholder` with the content of `memo-placeholder`.

Many Javascript APIs use `data-` tags to configure their rendering (for example grid APIs commonly use this to configure columns), and the above mechanism can be used to translate these attributes as well.

The API of the `Rtl.HTMLTranslate` unit is quite simple:

```
THTMLTagTranslator = class(TComponent)
Public
  procedure SetLanguageData(aData : TJSObject);
  Procedure TranslateHTMLTag(aEl : TJSHTMLElement; aScope : String = ''); overload;
  Procedure TranslateBelowHTMLTag(aEl : TJSHTMLElement; aScope : String = ''); overload;
  function GetMessageByName(const aScope, aName: string): String; overload;
  Function GetMessageByName(aRoot : TJSObject; const aScope,aName: string): String; overload;
  Function GetMessageByName(aScope : TJSObject; aName : string) : String; overload;
  Function GetMessageByName(aScope,aSeealsoScope : TJSObject; aName : string) : String; overl
  Function HasLanguage(aLanguage : String) : Boolean;
  Function GetScope(const aScope : String) : TJSObject; overload;
  Function GetScope(aRoot : TJSObject; const aScope : String) : TJSObject; overload;
  Property CurrLanguageStrings : TJSObject Read FCurrLanguageStrings;
Published
  Property DataTagName : String Read GetDataTagName Write FDataTagName;
  Property DefaultScope : String Read FDefaultScopeName Write FDefaultScopeName;
  Property ContinueKey : String Read FContinueKey Write FContinueKey;
  Property Language : String Read FLanguage Write SetLanguage;
  Property LanguageSource : TLanguageSource Read FFileMode Write FFileMode;
  Property LanguageFileURL : String Read FLanguageFileURL Write FLanguageFileURL;
  property LanguageVarName: String read FLanguageVarName write FLanguageVarName;
  Property TextMode : TTextMode Read FTextMode Write FTextMode;
  Property OnLanguageLoaded : TLanguageLoadedEvent;
  Property OnLanguageLoadError : TLanguageLoadErrorEvent;
end;
```

The class has the following public/published properties:

CurrLanguageStrings the data for the current language.

DataTagName Data attribute name. You can change this to use another data attribute. Default is 'translate'.

DefaultScope Default scope to use when looking for translations. First the scope passed in `TranslateHTMLTag` is checked, then the default scope, then `ContinueKey` is used.

ContinueKey Continue key name: When set, indicates the name of a scope in which to continue searching for a translation term. This can be used for form inheritance; `ContinueKey` can be set to continue searching in the inherited scope. By default it is empty.

Language Current Language name (will be lowercased)

LanguageSource Language source: All languages in a single file (lsSingle), languages reside in multiple files (lsMulti) or a variable in the Javascript global Scope (lsScoped) contains all languages.

LanguageFileURL The URL to the file to load language strings from. If `LanguageSource=lsMulti` this should be a format template with the language code is done (use for Example: `/lang-%s.json`.

LanguageVarName if `LanguageSource=lsScoped` then `LanguageVarName` is the name of the global `Window` property that contains the translations.

TextMode Textmode determines whether the translator uses `InnerText` (`tmText`) or `InnerHTML` (`tmHTML`) to set the translated text. Note that using `tmHTML` can result in a security leak as Javascript could be injected.

OnLanguageLoaded An event triggered when a language is loaded (whatever the mechanism).

OnLanguageLoadError An event called when an error occurs during loading of a language file.

The class has the following public methods:

SetLanguageData Directly set the language data object to `aData` instead of using an URL. The data will be interpreted according to `LanguageSource`.

TranslateHTMLTag Translate a single HTML tag `aEl`, using indicated scope `aScope`. If the scope is not given, the root scope of the current language is used or the default scope.

TranslateBelowHTMLTag Translate the HTML tag `aEl` and everything below it, using indicated scope `aScope`. If the scope is not given, the root scope of the current language is used.

GetMessageByName Search for a message named `aName` in the indicated `aScope`, optionally `aSeeAlsoScope`. In case the scope name is given you can start the search for the scope in `aRoot` The message is empty if no message was found.

HasLanguage check if the given language is available. In case of lsMulti, the current language must match the passed `aLanguage`.

GetScope find the scope object named `aScope`. if `aRoot` is given, start the search at `aRoot`.

Note that the notion of 'scopes' is used. This is done for use in SPA applications: each "page" in the SPA can use its own scope when looking for translations: this means that there is no need to invent globally unique names for the texts: names only need to be unique in a scope.

So, how to use this API to translate our application ? First of all, our HTML must be extended with the `data-translate` attribute:

```
<!doctype html>
<html lang="en">

  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="bulma.min.css">
    <title data-translate="Title">Translation using resource strings - Manual</title>
    <script src="translate.js"></script>
  </head>

  <body>
    <div class="section">
      <div class="box">
        <h2 class="title is-2"
            data-translate="Header">
            Translation using HTMLTranslator
        </h2>
        <p class="block"
           data-translate="Paragraph">
           This text will be translated using data-translate.
        </p>
        <div class="field is-horizontal">
          <div class="field-label is-normal">
            <label class="label"
                   data-translate="Name">
              Name
            </label>
          </div>
          <div class="field-body">
            <div class="field">
              <p class="control">
                <input class="input"
                       type="text"
                       placeholder="Type your name"
                       data-translate="Input-placeholder">
              </p>
            </div>
          </div>
        </div>
        <button
          id="btn-translate"
          class="button is-link"
          data-translate="Button">
          Translate this page
        </button>
```

```
      </div>
    </div>
    <script>
      window.onload=rtl.run;
    </script>
  </body>
  </html>
```

Secondly, we need to create a file with the translations. Note that, contrary to the resource string mechanism, all translations must be present, also the original language: there is no way to reset the translation.

The file looks as follows:

```
{
  "nl" : {
    "html" : {
      "Title":"Vertaling met HTMLTranslator",
      "Button":"Vertaal deze pagina",
      "Header":"Vertaling met resourcestrings",
      "Paragraph":"Deze tekst wordt vertaald.",
      "Input-placeholder":"Typ uw naam",
      "Name":"Naam"
    }
  },
  "en" : {
    "html" : {
      "Title" : "Translation using HTMLTranslator",
      "Header" : "Translation using HTMLTranslator",
      "Button": "Translate this page",
      "Paragraph" : "This text will be translated using data-translate",
      "Input-placeholder":"Type your name",
      "Name":"Name"
    }
  }
}
```

As you can see, we've elected to use the scope "html". We save this file as `lang.json`, next to our index.html page.

Lastly, we adapt the code. The program starts by initializing an instance of `THTMLTagTranslator`. It sets the default scope to `'html'` and set the `LanguageSource` to `lsMulti`, since we have multiple languages in our file.

The `LanguageFileURL` is set to `lang.json`, the name of the file we saved the translations in, it will be downloaded from the same location as the `index.html` page.

```
uses
  JS, Classes, SysUtils, Web, Rtl.HTMLTranslate;

Var
  aButton : TJSHTMLButtonELement;
  aTranslator : THTMLTagTranslator;
  IsDutch : Boolean;
// some code removed for clarity...
Procedure Init;
```

```
begin
  aTranslator:=THTMLTagTranslator.Create(nil);
  aTranslator.LanguageSource:=lsMulti;
  aTranslator.LanguageFileURL:='lang.json';
  aTranslator.OnLanguageLoaded:=@DoLangLoaded;
  aTranslator.DefaultScope:='html';
  aButton:=TJSHTMLButtonELement(Document.GetElementByID('btn-translate'));
  aButton.onclick:=@DoTranslation;
end;

begin
  Init;
end.
```

As you can see, the code to translate is quite simple. As soon as the language is set, the `DoLangLoaded` is triggered, and the `TranslateBelowHTMLTag` can be used to do the actual translation. Note that the code does not pass a scope to use to the `TranslateBelowHTMLTag` call: the default scope is set to `html`, and this is sufficient. Lastly, the `OnLanguageLoaded` is set to `DoLangLoaded`, where we do the actual translation:

```
procedure DoLangLoaded(Sender: TObject; aLanguage: String);
begin
  aTranslator.TranslateBelowHTMLTag(TJSHTMLElement(Document.body));
  aTranslator.TranslateBelowHTMLTag(TJSHTMLElement(Document.head));
end;
```

As you can see, we call the `TranslateBelowHTMLTag` method twice:

- once for the `body` element.
- once for the `head` element. This call will translate the title tag, thus setting the page title.

Note that we do not pass a scope name to the `TranslateBelowHTMLTag` call. This will cause the code to use the default scope, `'html'`.

The `onclick` handler of our button is now quite simple:

```
function DoTranslation(aEvent: TJSMouseEvent): boolean;

begin
  Result:=True;
  IsDutch:=Not IsDutch;
  if IsDutch then
    aTranslator.Language:='nl'
  else
    aTranslator.Language:='en'
end;
```

Setting the language will load the translation file if necessary, and when loaded, will call our `DoLangLoaded` event handler.

That's all there is to it.

4 Creating a file with translations

It is not necessary to create the JSON file completely by hand: the pas2js repository contains a tool that allows you to extract all texts that must be translated from the HTML files. The tool is called `extractlang`, it can be found under the `tools` directory of the pas2js distribution.

Currently, only a command-line tool is available. When running it with the `-h` command-line option, it displays a help page:

```
Usage: /home/michael/P2JS/main/tools/extractlang/extractlang [options]
Where options is one or more of:
-h --help                This help text
-c --clear               Clear output JSON file
                         (Default is to update existing file).
-d --html-dir=DIR        Directory with HTML files to scan
-f --file-mode=MODE      Set file mode: one of single or multiple
-o --output=FILE         File to write JSON translations
                         (may get suffix depending on file mode)
-l --languages=LIST      Comma-separated list of languages to create
-m --minify              Minify output
-n --name=NAME           Set name of data-tag to NAME (data-NAME)
-r --recurse             Recurse into subdirectories of the HTML directory
-s --single-scope=SCOPE  Put all translation names in a single scope
-t --trash-values        Trash values for other languages
```

To operate, the tool needs at least 2 options: `-d` and `-o`. The `-d` option specifies a directory where the tool will look for HTML files. The `-o` option specifies the name of a JSON language file to create or update.

With these options, the tool will scan the directory for HTML files (and subdirectories if `-r` is specified), collect all HTML tags with attribute `data-translate`, and will write the inner text of this tag in a JSON file. By default, only a `en` language will be created. But you can also create the text for additional languages using the `-l` option: for each language, the tool will check if the named text is there, and if not, it will add the text.

If we run the program in the directory of our sample application using the following command-line:

```
extractlang -d . -o lang.json -l 'en,nl' -s html
```

We specify the `-s` option because that's the scope we used in code. If no scope is specified, for each processed HTML file a separate scope will be used, where the scope name is the HTML filename, lowercased and without extension.

The tool will give you some diagnostic output:

```
Searching ./index.html for translatable terms, adding to scope : html
Found 6 translatable terms
Collected 1 message scopes
Copied 1 new scopes with 6 words, added 0 new words in existing scopes.
```

And we will end up with the following file:

```
{
```

```
  "en" : {
    "html" : {
      "Title" : "Translation using resource strings - Manual",
      "Header" : "Translation using HTMLTranslator",
      "Paragraph" : "This text will be translated using data-translate.",
      "Name" : "Name",
      "Input-placeholder" : "Type your name",
      "Button" : "Translate this page"
    }
  },
  "nl" : {
    "html" : {
      "Title" : "Translation using resource strings - Manual",
      "Header" : "Translation using HTMLTranslator",
      "Paragraph" : "This text will be translated using data-translate.",
      "Name" : "Name",
      "Input-placeholder" : "Type your name",
      "Button" : "Translate this page"
    }
  }
}
```

This file is ready to be translated. (note that if you use the minify option, the output is not human-readable and difficult to translate manually)

By default, the tool will load an existing file and will simply add new texts, which makes it suitable for updating existing translations.

As an aid in translating, you can specify the -t (trash) option. In that case, the first language will contain the original text from the HTML file, and all new words in other languages will contain some chinese characters. Doing so can aid as a visual check when changing the language in the actual HTML page: any non-translated terms will appear as Chinese characters which (for western people) will stand out.

5 Conclusion

Pas2js offers all tools needed to make a localized application. The mechanisms for translation of resource strings and HTML pages are similar in the sense that they use a JSON file: this allows you to have translations of both resourcestrings and HTML tags in a single JSON file.

As always, there is some room for improvement, such as integration in the Lazarus IDE, a GUI program for the extractor tool and a tool to manage translations. These will be created in time. But all the basic mechanisms are there, so the impatient developer can already get started.