Building visually consistent, multilingual websites with Apache and mod_perl

Javier A. Rodríguez
R&D Manager, LatinB2B
jrodriguez@latinb2b.com
September 6, 2000

Introduction

In the course of designing a large multilingual site, there are more challenges than providing a content management system or just making all pages look alike. Making browser, proxy and server work together to respond to user interaction, as expected, and deliver content to a multilingual, multicultural audience has been a challenge for Latin-American web designers for several years.

This article deals with some possible solutions to this problem using Apache and mod_perl, but the concepts shown here also apply to other web servers and other language interfaces to the Apache API

In the rapidly moving times of the Internet, it is unrealistic to believe that a given piece of information will stay relevant for too long once committed to non-digital means. For this reason, the latest version of this document is available online at http://people.latinb2b.com/~jrodriguez/

A bit of history

Multilingual support has been a necessity for Latin American web developers since the commercial acceptance of the web. Most companies would request a bilingual site to disseminate corporate information to both local and foreign customers, suppliers and investors. The simplest solution –which is used a lot even these days- is having the site split logically in as many languages as needed and presenting the user with a menu that lets the visitor choose an initial path depending on his preference. Even though this approach is simple and straightforward, it is rather unpractical for large sites since it imposes a single point of entry for the site, is an extra step to reach useful content, and forces the user to make a language choice every time he returns.

In addition, consider what happens when the user arrives to an “inner” page through a different site or a search engine. If the user needs to access the content in a different language, he must go back to the language selection menu and navigate back to the equivalent page in his preferred language – assuming that the user correctly guessed that there was a different version at all. This only gets worse while accessing historic archives or when the navigational system is confusing.

Overall, the problem of switching between languages can be addressed with a template-driven system that -besides providing a consistent look across the entire site- allows every page to include a language selection menu, but this doesn’t solve the problem of having the user make an explicit choice every time he comes back.

Even before the Accept-Language HTTP header existed (transmitted by browser for years) there were a few ways to make an educated guess about a user's native language, that is, inspecting his browser's User-Agent string or the visitor's domain name.

For instance, back in 1996, Version 2 of Netscape browsers would send an HTTP Accept-Language header when explicitly configured to do so. Even then, the same version in English would identify itself as "Mozilla/2.02 (Win95; I)", while a version in a foreign language would send a different User-Agent, i.e. "Mozilla/2.02 [es] (Win95; I)" for the version in Spanish. The problem was that a lot of Spanish-speaking users would use the English version of the browser because it was the version distributed or supported by their ISP, and mostly because there was a significant gap of time between the English release of a browser and the equivalent translated version. Unfortunately, this meant that English releases were less buggy than the older translated versions.

Inverse domain name resolution –though slow- could also give a clue of the visitor's language if mapped to one of the languages spoken in the users' country. For instance, a visitor accessing the site from a host with a top-level domain equal to '.mx' indicates that the user has a great possibility of being physically located in Mexico and that his native language might be Spanish. Of course, this approach fails miserably for countries with several official languages and .com and .net top-level domains, which are not geographically bound to a single country.

In short, none of these last two approaches was foolproof, but at that time it was better than nothing.

The challenge

LatinB2B was founded in October of 1999; as part of its value proposition we would build a brochureware site providing company information and a set of vertical industry portals offering access to local, premium quality information resources relevant to a given industry in each visitor's native language. All content should be available in the three languages most commonly used in Latin America: Spanish, Portuguese and English. The "look and feel" of every vertical industry portal should be consistent among all languages, and a visitor must be able to switch between languages if needed.

The solution

During the analysis phase of LatinB2B's portals, several choices had to be made. The choice of operating system and web serving software was an easy one, given our years of experience using open source software. After weighing code maturity, scalability, reliability and price, we decided to go for Apache under Linux.

Content management was an obvious choice after this. Integration of WebDAV and CVS tools gave us a powerful content management system that everyone could use through an easy-to-use graphical user interface.

Visual consistency was a non-issue, since a lot of template substitution systems had been developed for Apache, and it was rather trivial to integrate a SSI-like template-driven system using chained content handlers in mod_perl.

However, language selection was still an issue. The most desirable feature was auto detection of each visitor's preferred language using all available information.

Content negotiation

The obvious place to turn to was Apache's built-in content negotiation features.

The HTTP/1.0 spec in RFC1945 [5] defines the following two headers:

"D.2.4 Accept-Language

> "The Accept-Language request-header field is similar to Accept, but restricts the set of natural languages that are preferred as a response to the request.

"D.2.5 Content-Language

> "The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this may not be equivalent to all the languages used within the entity.

So in theory, the browser sends a list of language tags that indicate the user preference and with this information, the server decides which versions are available and decides which version to send, appropriately marking it.

These tags are defined in RFC1766 [4], and are composed of a primary language tag and an optional set of subtags. The primary language tags, consisting of exactly two letters, are interpreted according to the ISO 639 standard [2], which defines the codes for the representation of language names. The subtag may identify country or dialect information related to the language or even an unregistered language, but in browsers this usually identifies the country variant of a language and can be used to derive locale information.

For instance, let's say that a Netscape Communicator 4.75 browser is configured to present content, in order of preference, in English, Spanish with a Mexican locale, and French with a French locale. In that case, it will transmit an Accept-Language header as part of the request with the value "en, es-MX, pt-BR"; Internet Explorer 5, by the way, will transmit "en-us, es-mx;q=0.7, pt-br;q=0.3", in closer compliance with the RFC (!).

The value of this header will be available to CGI programs with the name transformed in the usual way (all caps, with dashes transformed to underscore chars and HTTP_ prefix). In this case, as the HTTP_ACCEPT_LANGUAGE environment variable.

Let's take a look at the CGI environment:

```
#!/usr/local/bin/perl

print "Content-type: text/html\n\n";
for(sort keys %ENV) {
    print "<B>$_</B>=$ENV{$_}<BR>\n";
}
```

Among a plethora of environment variables we will find something like the following:

```
HTTP_ACCEPT_LANGUAGE=en-us,es-mx;q=0.7,pt-br;q=0.3
```

For this feature to work properly, the user must configure his specific language preferences in his browser. The procedure varies from one browser to the next, but fortunately enough, the Debian Project has put together a comprehensive reference[19] to perform this configuration in the most popular browsers.

Here's a very important excerpt from this document:

> "One thing you need to be careful of is using sub-categories of languages. Using 'en-GB, fr', for example, does not do what most people expect (if they have not read the http specification). A server that receives a request for a document with a preferred language of 'en-GB, fr' when both an 'en' and 'fr' version exist will serve the French one. It will only serve the English document before the French one if there is a version of the file with en-gb for the language extension. Thus, you should configure your browser to send 'en-GB, en, fr' or simply 'en, fr'. It does work the other way though, e.g. a server can return en-us when en is requested.

We will see a practical example of this "feature" in the following section, but in the mean time please note that most users are unaware of this recommendation.

MultiViews

Multiviews are the simplest way to turn on content negotiation. IF the MultiViews option is present and the requested file is absent, Apache performs a pattern match on the filename to construct an internal Type Map, which is simply a list of potential files that fulfill the visitor's browser capabilities and language preferences.

To get Apache to serve multilingual content using this feature, add the following lines to the *httpd.conf* file:

1. If you compiled Apache yourself, make sure that the mod_negotiation module is active. If you got a binary version it is usually compiled in by default. As a quick check in Unix, the following two lines should appear near the top of the *httpd.conf* file.

```
LoadModule negotiation_module libexec/mod_negotiation.so
(…)
AddModule mod_negotiation.c
```

2. Map the ISO codes of the languages you will support to the extensions of the files you will use: i.e.:

```
AddLanguage en .en
AddLanguage es .es
AddLanguage pt .pt
AddLanguage fr .fr
AddLanguage de .de
```

3. Define the default priority of these languages:

```
LanguagePriority en es pt fr de
```

4. Define the MultiViews option inside a directory or location:

```
<Directory "/opt/apache/servers/apachecon/docs/multiviews">☒
Options MultiViews
DirectoryIndex index
</Directory>
```

As you can see, Apache can be configured to serve multilingual content in less than five minutes. Unfortunately, this has a price: a filename pattern match must be performed for every request.

Type maps

A type map explicitly lists all variants of a given URI and their attributes. It allows more sophisticated weighing of the file's characteristics depending on both the client preferences and an explicit quality factor for each file, which is meant to provide a guide

over the "best" file format as defined by the author. This feature is most useful when a given file is available in different formats and it's unknown if the client has support for all of them.

Consider for instance a documentation file available in RTF, HTML, PDF and PS, being the original file format RTF. Sample quality ratings for these files might be as follows:

File format	Quality factor
RTF	1.0
PostScript	0.8
PDF	0.5
HTML	0.3
Plain text	0.1

This means that the RTF format would be preferred over all the others, perhaps because it is the original version and can be easily edited and transformed into the others. The PostScript and PDF versions follow because they can be printed without giving up quality. The HTML version follows because it preserves some format attributes, and the plain text version is also available just in case the other ones are unacceptable.

Of course, this feature can also be used to negotiate content language as well. In this case, quality factors are also followed although concerning languages quality rating is not meaningful.

To actually use this feature in Apache, we need to add another configuration directive to *httpd.conf* (besides those in the last section):

```
AddHandler type-map var
```

This line tells Apache that it should treat filenames matching *filename*.var as the type map for that given file.

Now, for each file we will serve, we need to construct a type map. This file usually has this format:

```
URI: index.en.html
Content-type: text/html
Content-language: en,en-gb,en-us
Description: "English version (Original)"

URI: index.es.html
Content-type: text/html
Content-language: es,es-mx
Description: "Spanish version (translation)"

URI: index.pt.html
Content-type: text/html
Content-language: pt
Description: "Portuguese version (translation)"
```

Using this feature, if a visitor requests the document named 'index', the type map will be consulted and all possible variants will be weighed.

Perhaps the niftiest trick with explicit type maps is that if none of the versions found meet the client's requirements, an HTML menu describing all those available will be shown.

Disadvantages of mod_negotiation

Consider the following two scenarios of the use of mod_negotiation:

- The user has configured [es-AR, pt-BR, en-GB] as his preferred languages. Since none of the languages match, Apache will return the first available version of the document defined by the LanguagePriority directive given before.

Of course, this is the user's fault and not Apache's because he didn't set appropriate fallback values in the browser language preferences, but it is a serious problem nonetheless. We can work around this limitation using more AddLanguage directives and an unhealthy number of symlinks, but it becomes rather impractical as we add support to new languages.

- The user has configured his browser to use [en, es, pt] and while navigating the site he decides that he wants to see the content in another language. He fiddles with his browser preferences to set [es, en, pt] and hits the "Reload" button.

Probably nothing will happen: even tough Apache is intelligent enough to mark the negotiated docs as non-cacheable, most browsers think they know better and get the document back from their local cache, a huge mistake! In some cases, even if HTML documents load in the correct language, previously negotiated images would stay in the previous language. The caching problem only gets worse as we add more browsers, proxies and firewalls to the mix.

Additionally consider that when using explicit type maps they must be synchronized with the alternates. If you decide to take this approach, seriously consider writing a script to avoid maintaining the type maps by hand.

We see that even though Apache's content negotiation module is adequate for some tasks, it falls short for proper language selection. In short, a good language selection mechanism must determine a user's explicit language choice or otherwise establish a sensible default. The user must be able to change the language at any time and the user interface must change accordingly.

Using Apache::SelectLanguage

Once the user has made a choice, it should be remembered. Being HTTP a stateless protocol, there's no standard way to preserve state on the server. Luckily enough, the problem has been pondered before, and the most straightforward way to do that is to use a client-side cookie to store the language tag. Further elaboration on cookies and the way they work is beyond the scope of this article, but there are many articles and references about them on the Web [16][17][18].

Apache::SelectLanguage is a simple module that aids in determining the user's language, taking into account a variety of client information. It is available online at http://people.latinb2b.com/~jrodriguez/perl/.

Apache::SelectLanguage is meant to be installed as a PerlFixupHandler under Apache+mod_perl using –for instance- the following configuration directives on *httpd.conf*:

```
PerlModule Apache::SelectLanguage
PerlSetVar Languages "es,en,fr,de,pt"
PerlSetVar LanguageHandlers "cgi(_lang),cookie(language),accept"
PerlFixupHandler Apache::SelectLanguage
```

The `PerlSetVar Languages` directive lists all the languages that we intend to support in our site.

The `PerlSetVar LanguageHandlers` directive lists in direct execution order the handlers that will be `eval`'ed; each handler can receive an optional argument. If a given language handler fails to recognize a match between a language set as a user preference and one of the available languages, it will pass control to the next handler in line - very much like the standard Apache handler mechanism.

These are the language handlers available as of September 6, 2000:

Handler	Description
cgi(*varname*)	Specifies that the language should be extracted from the QUERY_STRING of a GET request. Internally, the value of the language is extracted from the request's `args` hash. `cgi(_lang)` will try to extract the language tag from the CGI variable named "_language"
cookie(*cookiename*)	Specifies that the language tag should be extracted from the cookie with the name given. `cookie(language)` will try to retrieve the language tag from the cookie named "language".
Accept	The language tag should be extracted from the Accept-Language HTTP header.
subprocess_env(*varname*)	The language tag should be extracted from the request's `subprocess_env` variable with the given name. This is meant to be used with a value set by a previous handler.

If none of the handlers returns a valid language, the first language specified in the `PerlSetVar Languages` variable will be taken as the default.

Once a language tag has been determined, a couple of environment variables are set with `$r->subprocess_env()`:

Environment variable	Description
LANGUAGE	the preferred language tag
LANGUAGE_SET	the name of the handler that made the determination

Arbitrary language handlers and response setup routines can be added through subclassing.

An added benefit of this approach is that the LANGUAGE environment variable usually has a meaningful use for the underlying OS. Consider this CGI:

```
#!/usr/local/bin/perl

use CGI;
use CGI::Carp qw(fatalsToBrowser);

my $file = "/etc/passwd";
open(FILE,">>$file")||die("$file: $!");
die("I'm in deep trouble");
```

When run under a Unix variant with several locales installed (i.e. Debian GNU/Linux) the `die()` clause will output an error message in the language set by Apache::SelectLanguage.

Other than that, the module is not very useful by itself. In order to serve meaningful content, it will be necessary to use in some way the language value it has returned. In the case of static content, the two modules listed below will complete the content delivery cycle in a meaningful way. And in the case of server programs in the form of CGIs or other content handlers (including programs running under Apache::Registry), it's best to use the LANGUAGE environment variable to indicate the desired output language. Instead of hardwiring messages in your code, i.e.

```
print("Invalid password");
```

you can use something like

```
my $lang=$ENV{LANGUAGE};
print(translate('MSG_INVALID_PASSWORD',$lang));
```

where `translate` is a custom message translation subroutine. In order to perform this translation of message names into meaningful phrases for each language, you can use either the GNU Gettext module for Perl, the Apache::Language module or a proprietary homebrew method.

Filename translation

Once the preferred user language has been selected, it is necessary to map the URI to a static file residing in the server's hard disk, independently of whether the file will have dynamic content attached. There are at least a couple of filename mappings that can help to simplify content management chores.

	Single directory hierarchy	One hierarchy per language
Filename mapping	*/directory/filename.language.ext*	*/language/directory/filename.ext*
Sample layout	about staff.en.html staff.es.html staff.pt.html media pr-20000321.en.html pr-20000321.es.html pr-20000321.pt.html	es about staff.html media pr-20000321.html en about staff.html media pr-20000321.html pt about staff.html media pr-20000321.html
Pros	Allows spotting missing files at a glance	Allows more flexible permission management
Cons	Permissions must be assigned carefully	Missing files are hard to spot
Recommendation	Best when supporting a small number of languages	Best for numerous staff or external translation services

The following listing shows a simple perl module that implements the "Single directory hierarchy" strategy

```
package MyPackage::Translate;

use strict;
use Apache::Constants qw(OK DECLINED);

sub handler {
        my $r = shift;
        return DECLINED unless $r->is_initial_req;

        my $lang = $r->subprocess_env('LANGUAGE');

        $r->uri(setext($r->uri,$lang));
        $r->filename(setext($r->filename,$lang));
        return OK;
}

sub setext {
        my $str = shift;
        my $ext = shift;
        $str=~s,(\..*?),.$ext$1,;
        return $str;
}

1;
```

MyPackage/Translate.pm

This module is also meant to be used as a PerlFixupHandler. To use it, create a directory named MyPackage under a directory in @INC and copy this file in it. Then add the following section to *httpd.conf*:

```
PerlFixupHandler Apache::SelectLanguage
PerlFixupHandler MyPackage::Translate
```

It is trivial to modify the Translate.pm module to implement the "One hierarchy per language" strategy; the actual implementation is left as an exercise to the reader.

Serving actual multilingual content

So far, we have just slightly improved mod_negotiate's language selection capabilities. If we restart Apache and load a few dummy files, we will see that the content is served correctly but still fails to change properly after selecting a different language.

The last part of the equation can be addressed by the following sample module:

```
package MyPackage::Deliver;

use strict;
use Apache::Constants qw(OK DECLINED MOVED);
use Apache::File;

sub handler {
        my $r = shift;
        return DECLINED unless $r->is_initial_req;

        my $filename = $r->filename;
     # Redirect to index.html on directory request
        if($r->content_type() eq 'httpd/unix-directory') {

                $r->content_type('text/html');
                $r->status(MOVED);
                my $uri = $r->uri();
                $uri.='/' unless ($uri=~m,/$,);
                $uri.='index.html';
                $r->err_header_out('URI', $uri);
                $r->err_header_out('Location', $uri);
                $r->send_http_header();
                return OK;
        }
        my $fh = Apache::File->new($filename) || return DECLINED;
        my $date = (stat $r->finfo)[9];
        $r->set_last_modified($date);
        my $lang = $r->subprocess_env('LANGUAGE');
        my $etag = "$date;$lang";
     $r->content_language($lang);
     # Mark explicitly as non-cacheable
        $r->no_cache(1);
        $r->send_http_header();
        return OK if $r->header_only;
        $r->send_fd($fh);
        return OK;
}

1;
```

MyPackage/Deliver.pm

This module will serve static content adding the Content-Language header and marking the file as non-cacheable.

Working around buggy caches

Having control over the content delivery using a custom Content Handler allows us to control an additional aspect of the server response: the caching behavior.

However, consider what happens when the user for whatever reason decides to change the language. Internet Explorer, for instance, will compare the URL of the page with the one in its cache and decide that it is the same document the user was viewing before, and will refuse to fetch the content from the server. Normally, a hard reload is needed to dissuade the browser from getting the document from the cache.

We need to make a tradeoff between marking every page as non-cacheable and leaving the decision to the browser.

	Default behavior	Marking content as Non-cacheable
Bandwidth	Page stays in browser cache	Page must be transmitted with every visit
Action upon language switch	URL must change to force reload	Page is reloaded anyway
Recommendation	Best for static content, including images	Best for dynamic content or static content that might change upon language negotiation

Of course, there is more than black and white. Depending on the application, you could decide that only files with .html or .txt extension are to be translated:

```
<FilesMatch "\.(html|txt)$">
       PerlFixupHandler Apache::SelectLanguage
       PerlFixupHandler MyPackage::Translate
</FilesMatch>
```

In this case, you should make sure that references to files with different extensions include the language tag. Apache would transmit HTML and plaintext files every time they are requested, but it will preserve the normal cache behavior for images and other inline content. This is not an issue since the images for files in different languages have different URIs. This mixed approach represents an acceptable compromise and may be appropriate for most purposes.

Now if content is effectively static, you want to make sure that each page is transmitted only once for each request in each language and you enjoy living on the edge, you can use entity tags (ETags) to mark each page uniquely in the browser's cache. Take into account that this is an HTTP/1.1 feature and its functionality might be broken or even totally absent in pre-4.x browsers and some proxies, leading to unpredictable results.

To enable this feature, delete the lines reading

```
      # Mark explicitly as non-cacheable
         $r->no_cache(1);
```

and substitute them for this:

```
      # Send Entity Tag
         $r->header_out('Etag'=>$etag);
```

More info on the ETag is available on the HTTP/1.1 spec and on Stas Bekman's most excellent mod_perl guide in the section about Issuing Correct HTTP Headers and Entity Tags. However, let me advance a word of caution: don't use entity tags on files with dynamic content.

In any case, a nifty trick to force document reload implies modifying the URL of the actual request appending a query string. If the content is a CGI, care should be taken to modify the query string instead of appending to it. Note that this technique will not work on a page that was loaded using the POST request.

A simple template system

The following module is a modification of MyPackage::Deliver that just adds a static header and footer to each file request, providing the basis to show a consistent look across the entire site.

```
package MyPackage::Deliver2;

use strict;
use Apache::Constants qw(OK DECLINED MOVED);
use Apache::File;

sub handler {
        my $r = shift;
        return DECLINED unless $r->is_initial_req;

        my $filename = $r->filename;
        if($r->content_type() eq 'httpd/unix-directory') {

                $r->content_type('text/html');
                $r->status(MOVED);
                my $uri = $r->uri();
                $uri.='/' unless ($uri=~m,/$,);
                $uri.='index.html';
                $r->err_header_out('URI', $uri);
                $r->err_header_out('Location', $uri);
                $r->send_http_header();
                return OK;
        }
        my($fh, $fh2);
        $fh = Apache::File->new($filename) || return DECLINED;
        my $date = (stat $r->finfo)[9];
        $r->set_last_modified($date);
        my $lang = $r->subprocess_env('LANGUAGE');
        my $etag = "$date;$lang";
        $r->content_language($lang);
        $r->header_out('ETag'=>$etag);
        $r->send_http_header();
        return OK if $r->header_only;
        $fh2 = Apache::File->new(setext($r-
>dir_config('DeliverHeader'),$lang));
        $r->send_fd($fh2);
        $r->send_fd($fh);
        $fh2 = Apache::File->new(setext($r-
>dir_config('DeliverFooter'),$lang));
        $r->send_fd($fh2);
        return OK;
}

sub setext {
        my $str = shift;
        my $ext = shift;
        $str=~s,(\..*?),.$ext$1,;
        return $str;
}


1;
```

In order to use it properly, the following lines should be present on *httpd.conf*:

```
PerlHandler MyPackage::Deliver2
PerlSetVar DeliverHeader /path/to/file/header.html
PerlSetVar DeliverFooter /path/to/file/footer.html
```

There are more complex, solid ways to put together a template system. In LatinB2B we use a third-party full content-management system called Xtra to display the customized vertical industry portals, but beyond the home pages most static and dynamic content is displayed through a chained content handler not unlike those discussed in Chapter 4 of Stein and MacEachern's book[20]. <plug shameless=1>Even if you are not interested in the minutiae of template-driven websites, if you are going to be using mod_perl at all you really want to get this book</plug>.

Putting it all together

The following code snippet will set a cookie with a default language for Apache::SelectLanguage. Note that in order to work around the buggy cache problem, it will redirect the browser to the referrer page adding a query string –or modifying an existing one. Besides that, the cookie will have an expiration time of one year so that the user does not need to reset the language when he comes back.

```
#!/usr/local/bin/perl

use Apache;
use Apache::Constants qw(:response);
use CGI;

use vars qw($r $cgi);

$r = Apache->request;
$cgi = CGI->new();

my @LANG = split(/,\s*/, $r->dir_config('Languages'));
my %args = $r->args();
my $lang = $args{'lang'};
if(defined($lang)) {
        if($lang) {
                $r->err_header_out('Set-Cookie', $cgi->cookie(-
name=>'language', -value=>$lang, -path=>'/', -expires=>'+1y'));
        } else {
                $r->err_header_out('Set-Cookie', $cgi->cookie(-
name=>'language', -value=>'x', -path=>'/', -expires=>'+1y'));
        }
        my $uri = $args{'uri'};
        $uri||='/';
        my $unique = int(rand()*0xffffffff);
        if($uri=~/\?/) {
                if($uri=~/_unique=/) {
                        $uri.=~s/_unique=\d+/_unique=$unique/;
                } else {
                        $uri."&_unique=$unique";
                }
        } else {
                $uri."?_unique=$unique";
        }
        $r->status(MOVED);
        $r->err_header_out(URI=>$uri);
        $r->err_header_out('Location'=>$uri);
        $r->send_http_header();
} else {
$r->content_type('text/html');
$r->send_http_header();
```

```
print "Languages are ",join(':',@LANG);
```

```
%LANG = (
        '' => 'Auto',
        'en' => 'English',
        'es' => 'Español',
);

my $referer=$ENV{HTTP_REFERER};
print $cgi->start_form(-method=>'GET');
print $cgi->popup_menu(-name=>'lang', -values=>['',@LANG], -
labels=>\%LANG, -default=>$ENV{LANGUAGE}, -onChange=>'form.submit()');
print $cgi->hidden(-name=>'uri',-value=>$referer);
print $cgi->end_form();

}
```

language.pl

In order to be used, the program must be installed under Apache::Registry.

There's always room for improvement

This brief article summarizes my experiences on incorporating multilingual support in a large website. I hope this humble recollection of code, tips and pointers can make someone else's job even a bit easier. In case you haven't noticed this article is far from being a definitive work on multilingual content serving, so I will appreciate any feedback on the points raised here and –more importantly- on the following points that are left completely open:

Multilingual search engines

I do not know what methodology is used by search engines to determine the language of a document, or if they apply the Varies HTTP header (see below) to index all language variants of a document. However, I am in favor of using the LANG attribute of the HTML tag as defined in the W3's HTML 4 recommendation to make known the language of all documents.

Varies HTTP header

I am not aware of a browser that uses the content on the HTTP Varies header. However a system using the recommendations in this document can be improved to be fully compliant with the HTTP/1.1 spec.

Apache 2.0 support

Even though I have played a little with Apache 2.0 betas, I ignore completely if there are going to be any changes on the underlying architecture and algorithms that might affect multilingual content serving and/or render this entire document as irrelevant by the time of a final release.

Conclusion

Following closely Perl's mantra "There's more than one way to do it", there are several working approaches to multilingual content serving. When implementing one of the techniques mentioned in this document, be sure to test it in as many browsers as possible, as there are plenty implementation bugs in browsers, caches and even in server software waiting to bite one of your users. Once you've finished testing, test again.

If you think you have tested enough, drop me a line and let me know how your project turns out, and if this document has been useful at all to you.

Bibliography and other resources

Relevant Standards

[1] HTML 4.01 Specification - Language information and text direction
World Wide Web Consortium
http://www.w3.org/TR/REC-html40/struct/dirlang.html

[2] Code for the Representation of the Names of Languages. From ISO 639, revised 1989.
Prepared by Robin Cover
http://www.oasis-open.org/cover/iso639a.html

[3] ISO 639:1988 (E/F). Code for the Representation of Names of Languages. First edition, 1988-04-01.
International Organization for Standardization (ISO).
Reference number: ISO 639:1988 (E/F).
Geneva: International Organization for Standardization, 1988.
iii + 17 pages.

Relevant RFCs

[4] RFC1766
Tags for the Identification of Languages
http://www.faqs.org/rfcs/rfc1766.html

[5] RFC 1945
Hypertext Transfer Protocol -- HTTP/1.0
http://www.faqs.org/rfcs/rfc1945.html

[6] RFC2068
Hypertext Transfer Protocol -- HTTP/1.1
http://www.faqs.org/rfcs/rfc2068.html

[7] RFC2070
Internationalization of the Hypertext Markup Language
http://www.faqs.org/rfcs/rfc2070.html

[8] RFC2231
MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations
http://www.faqs.org/rfcs/rfc2231.html

[9] RFC2277
IETF Policy on Character Sets and Languages
http://www.faqs.org/rfcs/rfc2277.html

[10] RFC2295
Transparent Content Negotiation in HTTP
http://www.faqs.org/rfcs/rfc2295.html

Internationalization and Localization

[11] perllocale(1)
http://www.perl.com/CPAN-local/doc/manual/html/pod/perllocale.html

[12] Apache::Language module
http://www.CPAN.org/modules/by-module/Apache/

[13] Locale::gettext module
http://www.perl.com/CPAN-local/authors/id/PVANDRY/

[14] Internationalization Reference List
Compiled by Eugene Dorr
ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/i18n-books.txt

[15] Techniques for multilingual Web sites
Jukka Korpela
http://www.hut.fi/u/jkorpela/multi/

Cookies

[16] Persistent client state – HTTP cookies
Preliminary specification
Netscape Corp.
http://home.netscape.com/newsref/std/cookie_spec.html

[17] RFC 2109
HTTP State Management Mechanism
http://www.faqs.org/rfcs/rfc2109.html

[18] Cookie Central
http://www.cookiecentral.com/

General resources

[19] Information on Pages Available in Multiple Languages
Debian Project
http://www.debian.org/intro/cn

[20] Writing Apache Modules with Perl and C
Lincoln Stein & Dough MacEachern
Home site at http://www.modperl.com/
Available online from O'Reilly and Amazon.com