

ApacheCon US
March 22, 2001
Singapore

**Tutorial: Multilingual sites with mod_perl and
Template Toolkit**

by **Stas Bekman**
<http://stason.org/>
<stas@stason.org>
eXtropia.com, Senior Engineer

This talk is available from: <http://stason.org/talks/>

This document is originally written in **POD**, converted to **HTML**, **PostScript** and **PDF** by
Pod::HtmlPsPdf Perl module.

(you will find a Table of Contents at the end of the Tutorial)

1 Multi-lingual Sites with Perl

1.1 Agenda

We are going to discuss the following issues

- Why it's almost always better to develop a dynamic site and not a static one, even if at the end you need it to be static.
- How do we figure out what language should be used to present the content.
- How do we store the language preferences once we know them.
- What major parts the generated output is generated from.
- What's the difference between site browsing and site searching.
- How the language specific data is to be stored in the database.
- Do we prepare a template per language, or keeping all languages in the same template?
- The convention of presenting dates and times vary from country to country. How do we address this issue.
- Finally we will talk about how to tell client browsers to render the output using a correct language encoding.

1.2 Dynamic versus Static Pages

Before you go and search for a solution for your multi-lingual enterprise, you have to figure out what kind of service you are going to provide: dynamic or static.

If the service pages are static, you have then to think whether you there will be many pages to maintain or just a few.

If you have only a few pages, I suppose the easiest solution is just prepare each page in each language and forget about it.

If you have many pages, it's pretty much the same whether your pages are dynamic or static, since the manual maintenance of many pages will be too time consuming, too error-prone, in other words -- ineffective. Therefore the correct solution is to approach the problem if it was a dynamic site, and generate the static pages using it. I will jump ahead and tell that Template Toolkit has a special program which generates a static site from dynamically generated output.

Therefore from now on we assume that you want to develop a dynamic site.

1.2.1 User Language Detection

Another important question is the process of figuring out what language should be used when presenting the content. The following algorithm tries to answer this question:

First, we separate users into two groups: those who visit the site for the first time, and those who have visited the site previously.

If you use cookies to track users (or some other mechanism that stores the information on the client side), and a user connects from the same machine/account he used when previously accessing your service, you should already know the language preferences: this bit of information can be stored in the cookie. This answers the question of language detection for the second group of users.

If you don't use cookies or some other mechanism to track users, it probably doesn't matter whether users have accessed the site beforehand or not, because you have no way to tell what their language preferences are.

Of course if a user has registered with your service, after logging in, you will know the language preference, since you can store it on the server side. But the problem is that even to display the login form, you need to know the user's language.

You can make an assumption about the language spoken by all of your potential users, and if you figure out that all of them understand (and can read) a common language, you don't need a multi-lingual site.

You can try to figure out the user's language by deducing it from his country. One way to determine the user's country is by doing a reverse DNS lookup on his IP address. This yields the user's computer name. You can then use of the TLD (top level domain) to make some reasonable assumptions about the language: chances are most users in the .fr domain can read French, for example. Since many hosts do not have correct reverse DNS mapping, you might also be tempted to deduce the country from the IP address itself. However this approach is due to fail in many cases: there are plenty of users whose visible IP address is outside their country. AOL users worldwide use AOL proxies located in the United States,

So we are coming back to the first group of users, whom we don't know about.

We must provide them a way to choose a language. The best way is to present a page with all available languages. Of course language name should be written in that language. All should be linked to the version of the service whose content is presented in that language.

We can go a little bit further and try to make an intelligent guess of the preferred language. This guess is made by looking at the `Accept-Language` header sent by most browsers. Localized versions of modern browsers set up the preferred language at install time. If the user knows that it's possible to adjust the language preference in her browser, there is a chance that she will. For example she might set the following preferences:

```
German
English-US
French
```

which might mean the following: My preferred language is German. I also understand American English to some extent and I know a little bit of French. (Of course a user might know all three languages perfectly, but still prefer one language over another).

When a browser sends a request to a server, it generates the following header:

`Accept-Language: de,en-us;q=0.7,fr;q=0.3`

where each language is separated by a comma. In some browsers, the preference level can also be specified for alternate languages. So in this example, I've marked American English as 70% and French as 30%.

You can either parse this header manually, but a better approach is to use standard CPAN modules. If you are using `mod_perl`, you can use the `Apache::Language` module, otherwise use the all-purpose `HTTP::Negotiate` module bundled into the `libwww` distribution.

Browsers bundled with OS or ISP packages are usually preconfigured with the language of the country the package was issued in. So if users aren't computer savvy, chances are that the default language setting will be correct.

If you have accepted this header, you may want to try your luck and present the top of the first page using the language derived from the header. But you still have to give the user an option to change the language, since the browser setting might be incorrect for this particular user.

Remember that `Accept-Language` is useful to make your service more user friendly and spare the hassle of picking the right language, but it doesn't come as a replacement for the standard way of presenting the available languages.

At this point we know the user's preferred language. In the case of a dynamic site, we proceed with generating the content. Otherwise we simply direct the user to the right static content.

1.3 Generating the Content.

When dynamic content is generated, at least two basic ingredients are used:

- the invariant data: like table and page headers which are always the same.
- the variant data: data which is not known a priori, and depends on user or some other input.

When a site is generated in a single language, these two items are easily implemented: Either use templates for invariant data or hard-code it in the code, and retrieve the variant data from database or other method.

When the requirements include multi-lingualism, these tasks become more complex. I'm going to talk about each of them separately.

1.4 Fetching Dynamic Data

We separate the dynamic data requests into two groups: those that require user input and those that don't.

A site search feature falls into into the first category, whereas browsing the site belongs to the second one.

1.4.1 Searching

Let's use a movie server and a user whose preferred language is French as an example. Our user searches a movie by entering search keywords in a search box.

If you have ever seen a text written in French, you know that it includes accented characters. This is not unique to French, but typical to many other languages. An accented character usually uses some non-accented character as a base. For example characters: *â, à, á* are all based on character *a*.

Since not all software supports accented characters or if only *us* keymap is available, a user might generate an input using only the base characters available in *us* keymap. In fact even with proper software and hardware support, most French users will type keywords with no accents. The server is still supposed to interpret this input correctly as if the accented characters were used.

We cannot guess which characters were inserted incorrectly, therefore the obvious solution is to make the search index free of accented characters. This means that you'll have to keep two versions of the text, one version adjusted for the search, and the original unaltered version. You need the original one, since you still have to output the correct text, regardless of the user's input limitations.

In this presentation I'll use ISO-8859-1 character set, which is used by many Western European languages.

The following code allows you to convert accented characters into their base characters:

```

package My::Language;

my %iso_8859_1_accents = (
  a => [ qw(à á â ã ä å Æ Á Â Ã Ä Å) ],
  c => [ qw(ç Ç) ],
  e => [ qw(è é ê ë È É Ê Ë) ],
  i => [ qw(ì í î ï Ì Í Î Ï) ],
  n => [ qw(ñ Ñ) ],
  o => [ qw(ò ó ô õ ö ø Ò Ó Ô Õ Ö Ø) ],
  u => [ qw(ù ú û ü Û Ü Û Ü) ],
  y => [ qw(ÿ Ý) ],
);

# build translation strings
my (%in, %out);
for my $letter ('a'..'z') {
  my $uletter = CORE::uc $letter;
  # translate non-accented letters
  $in{uc} .= $letter;
  $out{lc} .= $letter;
  $in{lc} .= $uletter;
  $out{uc} .= $uletter;
  if (my $ra_accented = $iso_8859_1_accents{$letter}) {
    my $in = join ' ', @$ra_accented;
    $in{lc} .= $in;
    $in{uc} .= $in;
    $out{lc} .= $letter x @$ra_accented;
    $out{uc} .= $uletter x @$ra_accented;
  }
}

# build translation subroutines
for my $type (qw(lc uc)) {
  my $sub = qq!
    sub iso_8859_1_$type {
      (my \$s = shift) =~ tr/$in{$type}/$out{$type}/;
      \$s
    }
  !;
  eval $sub;
}

# character sets
my %charsets = (
  'iso-8859-1' => { lc => \&iso_8859_1_lc,
                   uc => \&iso_8859_1_uc,
                 },
);

```

The code generates two methods: `iso_8859_1_lc()` which turns any input using ISO-8858-1 into a lowercase, accent-free version. `iso_8859_1_uc()` is similar but upper cases the input string.

For example when you call:

```
$stripped_lc = $charsets{'iso-8859-1'}{lc}->('Bienvenüe');
```

`$stripped_lc` will be set to:

```
bienvenue
```

These functions are used twice: First, when creating the search index and second, when accepting the search string, before the actual search is performed. Usually using the lower case for searching is the accepted technique.

In addition these functions can be used for sorting. Consider the following function:

```
sub cmp_nocase{
    return My::Language::lc($_[0], $_[1])
        cmp
        My::Language::lc($_[0], $_[2]);
}
```

which then can be used as:

```
my @correctly_sorted =
    sort { My::Language::cmp_nocase($lang,$a,$b) } @data;
```

where `$lang` is the currently used language (e.g. *fr*).

1.4.2 Browsing

When a user browses the site, preset data inputs are used (beware of the possibility for the user to change these inputs falsely assumed to be non- changeable). For example after a search has successfully completed and matches one or more records, you may list all the matched results, or a subset of them. From now on, the user clicks on one of the links to get to the full record.

At this point you may want to use the original text version, using all the characters, but... these should be encoded, since when the link is clicked it's possible that the browser will interpret the request incorrectly. To accomplish this you can use `URI::Escape::uri_escape()` or `Apache::Util::escape_uri()` (a much faster implementation under `mod_perl`).

Of course the text itself probably should be encoded, so the browser will not mess it up. The `HTML::Entities::encode()` or `Apache::Util::escape_html()` functions can be used for that.

1.4.3 Data Retrieval

Of course one of the big questions is how to build your database, so it will accommodate the multi-lingual capability. Obviously you should avoid creating language specific fields in every table that includes multi-lingual data. For example:

```
table movies:
-----
title_fr
title_en
title_es
description_fr
description_en
description_es
....
```

is a bad idea, because as you can see, the table will require many columns. Don't forget that the number of columns will actually be doubled, since you need to duplicate all the columns for the searchable version of the text). Every time you want to support a new language you'll have to alter the table and add many columns. Therefore it's better to place all the language specific data into one table:

```
id
lang
real_text
search_text
```

where *lang* specifies the language, *real_text* the real text, and *search_text* holds the version of the text adjusted for search. *id* is needed to map every record into the table the data belongs to.

Of course the *id* field is usually more complicated and might be comprised from additional fields. Sometimes you don't want to search all fields, but only specific ones. In one of the projects we have used three fields to represent a unique id:

```
orig_table
orig_column
orig_id
```

The concatenation of these three fields gave us a unique mapping from a record in the language specific data to the table it belongs to. For example:

```
SELECT * FROM lang where orig_table='movies'
AND orig_column='description' AND lang='fr'
AND search_text LIKE '%foo%'
```

will search only the description columns in the *movies* table. But if we want to retrieve all the language fields tied to some record we can do:

```
SELECT * FROM lang where orig_table='movies'
AND orig_id=123456
```

If you find out that your database implementation cannot cope with all the textual data in all languages in one table, you may want to consider to have a table for each language.

1.5 Invariant Data

Finally let's talk about invariant data. Data which doesn't change is either hard-coded in the code or, better, placed in a template.

Let's take for example a search feature. The template will look something like this:

```

<!-- results set -->
[% IF input %]
  [% IF total_hits %]
    A total of [% total_hits %] movies was found.
  [% ELSE %]
    No Results.
  [% END %]
[% END %]
<!-- input form -->
<form>
<input type="text"   name="input"   value="[% input %]" size="32">
<input type="submit" name="search" value="Search">
</form>

```

and a simple mod_perl script which will parse this template and produce the output:

```

use Template;
my $r = shift;
$r->send_http_header('text/html');
my $t = Template->new(INCLUDE_PATH => '/templates/path');
$t->process('search.ttml',
  { input      => 'foo',
    total_hits => 15,
  },
  $r
) or die $t->error();

```

So these are the template and code used in a single-language site. When we come to implement the multi-lingual site, we stand before this question: Should we have a page per language, or one page for all languages?

If you decide to go with the first option, you'll end up with many templates. Keeping them synchronized will be a nightmare. Therefore a much better approach is to keep all languages in one file. Whenever you add a new language or modify something, it's easy to do because all strings are stored in the same place.

We have to find a way to parse this file and extract only the text in the requested language. Therefore we've chosen to use XML tags, which will be then parsed by Template Toolkit so that texts in the right language will be selected.

We have used tag <text> for the text sections, and two letter code tags for language specific sections.

When applying these definitions the search input template becomes:

```

<form>
<input type="text"   name="input"   value="[% input %]" size="32">
<input type="submit" name="search"
  value="<text>
          <en>Search</en>
          <fr>Chercher</fr>
          <it>Cercare</it>
        </text>">
</form>

```

and the search results output template now looks like this:

```
[% IF input %]
  <text>
    <en>Search Results</en>
    <fr>Résultats de la recherche</fr>
    <it>Risultati della ricerca</it>
  </text>
  [% IF total_hits %]
    <text>
      <en>A total of [% total_hits %] movies was found.</en>
      <fr>[% total_hits %]
        [% IF total_hits > 1 %]
          films trouvés
        [% ELSE %]
          film trouvé
        [% END %]</fr>
      <it>[% total_hits %]
        [% IF total_hits > 1 %]
          movies trovati
        [% ELSE %]
          movie trovato
        [% END %]</it>
    </text>
  [% ELSE %]
    <text>
      <en>No Results.</en>
      <fr>Aucun résultat.</fr>
      <it>Nessun risultato.</it>
    </text>
  [% END %]
[% END %]
```

Of course it is almost impossible to edit this kind of template in WYSIWYG editors, but usually professional HTML designers use plain text editors, since today few WYSIWYG editors do a job of the same quality as one done by hand-coding HTML.

Template Toolkit allows us to adjust its template parsing methods, so we can plug in our own code. In our case we want it to pick only the text in the right language.

The following module overrides the default Template toolkit parsing method:

```

package My::Template::Parser;
require 5.005;

use strict;
use base qw(Template::Parser);

use constant LANG_RE      => qr{<([a-z]{2})>(.*?)</\1>}s;

#####
# constructor
sub new {
    my ($class, $options) = @_;
    my $self = $class->SUPER::new();
    $self->init($options);
    return $self;
}
#####
sub init {
    my ($self, $options) = @_;
    $self->{$_} = $options->{$_} for keys %$options;
}

#####
sub parse {
    my ($self, $text) = @_;

    # tokenize
    $self->_tokenize($text);

    # replace C<text> language sections with Template Toolkit
    # directives which will pick up the correct language text
    # at processing time.
    $text = '';
    for my $section (@{$self->{sections}}) {
        my $translated = $section->{text};
        $translated =~ s{@{[LANG_RE]}}
            {\[% IF lang=='$1' %\}$2\[% END %\]}gs
            if $section->{lang};
        $text .= $translated;
    }
    my $doc = $self->SUPER::parse ($text);
    return $doc;
}
#####
sub get_language_sections {
    return $_[0]->{sections};
}

```

```
#####
sub _tokenize {
    my ($self, $text) = @_ ;
    return unless defined $text && length $text;

    # extract all sections from the text
    $self->{sections} = [];
    while ($text =~ s!
        ^(.*)?                # $1 - start of line up to start tag
        (?
            <text                # start of tag
              (?:\s+id="?(\\d+)"?)? # $2 - optional id attribute
            >
              (.*)?             # $3 - tag contents
            </text>             # end of tag
        )
        !!sx) {
        push @{$self->{sections}}, { text => $1 } if $1;
        push @{$self->{sections}}, { lang => 1, id => $2||'', text => $3||'' }
            if defined $3;
    }
    push @{$self->{sections}}, { text => $text } if $text;
}
#####
1;
__END__
```

Note that you have to pass the *lang* variable to the template object, for this parser to do the right thing.

Then when you use the Template Toolkit to parse the template and generate the output, and assuming that the number of hits and the language are known, the following code is to be used:

```
use Template;
use My::Template::Parser ();
my $r = shift;
$r->send_http_header('text/html; charset=ISO-8859-1');
my $t = Template->new();
$t->_init(PARSER => My::Template::Parser->new,
        INCLUDE_PATH => '/search/path',
        ) or die $t->error();
$t->process('search.ttml',
        { input => 'foo',
          total_hits => 15,
          lang => 'fr',
        },
        $r
        ) or die $t->error();
```

So as you can see an overloaded Template parser is used to handle our customized template.

1.6 Handling Dates and Time Presentation

When presenting dates it doesn't make sense to keep translated date specific information in the database, since the data set is small and constant.

In addition different countries have different conventions for dates and time presentation.

So if the current date is Thursday March 22, 2001 and the time is 14:25 -- An American user will expect to read:

```
Thursday March 22, 2001 2:25pm
```

whereas a French user will expect:

```
Jeudi 22 Mars 2001 14h25
```

In this talk we will assume that these conventions are tied to languages, rather than countries. This is incorrect in reality, but this assumption is sufficient enough to be used as an example. In reality you may want to tie the conventions to countries and not languages. In this case you would need to ask the user her country preferences.

We specify the following data set for each language:

```
# all languages
%languages = (
  en => { name      => 'english',
         charset    => 'iso-8859-1',
         months     => [qw(January February March April May
                           June July August September October
                           November December)],
         days       => [qw(Sunday Monday Tuesday Wednesday
                           Thursday Friday Saturday)],
         date       => {short => '%m/%d/%Y',
                       medium => '%B %e, %Y',
                       long  => '%A %B %e, %Y',
                       },
         time       => {short => '%H:%M',
                       },
  },
  it => { name      => 'italiano',
         charset    => 'iso-8859-1',
         months     => [qw(gennaio febbraio marzo aprile maggio
                           giugno luglio agosto settembre ottobre
                           novembre dicembre)],
         days       => [qw(domenica lunedì martedì mercoledì
                           giovedì venerdì sabato)],
         date       => {short => '%d/%m/%Y',
                       medium => '%e %B %Y',
                       long  => '%A %e %B %Y',
                       },
         time       => {short => '%H:%M',
                       },
  },
},
```

```

fr => { name      => 'français',
        charset  => 'iso-8859-1',
        months   => [qw(Janvier Février Mars Avril Mai
                        Juin Juillet Août Septembre Octobre
                        Novembre Décembre)],
        days     => [qw(Dimanche Lundi Mardi Mercredi
                        Jeudi Vendredi Samedi)],
        date     => {short => '%d/%m/%Y',
                    medium => '%e %B %Y',
                    long  => '%A %e %B %Y',
                    },
        time     => {short => '%Hh%M',
                    },
},
es => { name      => 'español',
        charset  => 'iso-8859-1',
        months   => [qw(Enero Febrero Marte Abril Maio Junio
                        Julio Agosto Septiembre Octubre Noviembre Diciembre)],
        days     => [qw(Domingo Lunes Martes Miércoles Jueves Viernes Sábado)],
        date     => {short => '%d/%m/%Y',
                    medium => '%e %B %Y',
                    long  => '%A %e %B %Y',
                    },
        time     => {short => '%Hh%M',
                    },
},
);

```

Then we use this data to produce the dates and times in the correct language using the correct format.

These are handy compile time constants which are used in the date and time generators:

```

use enum qw(YEAR MONTH DAY);
use enum qw(HOUR MINUTE);

```

These are useful macros (implemented as callbacks) used in the formats above (they are derived from the format used by the `strftime(3)` function):

```
#####
# strftime compatible specifiers
# %A    full weekday name
# %B    full month name
# %d    day of the month (01-31)
# %e    day of the month (1-31)
# %H    hour (00-23)
# %I    hour, 12-hour clock (01-12)
# %k    hour (0-23)
# %l    hour, 12-hour clock (1-12)
# %M    minute (00-59)
# %m    month (01-12)
# %p    AM/PM
# %Y    year with century
# %y    year without century (00-99)
use vars qw(%strftime);
%strftime =
(
  A => sub { @{$languages{$_[1]}{days}}|[1]{week_day($_[0])} },
  B => sub { @{$languages{$_[1]}{months}}|[1]{$_[0][MONTH]-1} },
  d => sub { sprintf('%02d',$_[0][DAY]) },
  e => sub { sprintf('%d',$_[0][DAY]) },
  H => sub { sprintf('%02d',$_[0][HOUR]) },
  I => sub { sprintf('%2d',$_[0][HOUR]?$_[0][HOUR]>12?$_[0][HOUR]-12:$_[0][HOUR]:12) },
  k => sub { sprintf('%d',$_[0][HOUR]) },
  l => sub { $_[0][HOUR]?$_[0][HOUR]>12?$_[0][HOUR]-12:$_[0][HOUR]:12 },
  M => sub { sprintf('%02d',$_[0][MINUTE]) },
  m => sub { sprintf('%02d',$_[0][MONTH]) },
  p => sub { $_[0][HOUR] && $_[0][HOUR]<13 ? 'AM' : 'PM' },
  Y => sub { sprintf('%04d',$_[0][YEAR]) },
  y => sub { $_[0][YEAR]%100 },
);
```

These are two functions that show that accept either current date or time, the language and the requested format:

```

# $time_str = format_time($lang,$format_type,$time)
#####
sub format_time{
    my ($lang,$format_type,$time) = @_;
    # either can be zero
    return '' unless defined $time->[HOUR] and defined $time->[MINUTE];

    my $format = $languages{$lang}{time}{$format_type};
    warn("unknown time format: $format"), return '' unless $format;

    my $result;
    for my $chunk (split /(\%.)/, $format) {
        $result .= $chunk =~ /^%\(.)/ ? strftime{$1}->($time,$lang) : $chunk;
    }
    return $result;
}

# $date_str = format_date($lang,$format_type,$date)
#####
sub format_date{
    my ($lang,$format_type,$date) = @_;

    return '' unless $date->[YEAR] and $date->[MONTH] and $date->[DAY];
    my $format = $languages{$lang}{date}{$format_type};
    warn("unknown date format: $format"), return '' unless $format;

    my $result;
    for my $chunk (split /(\%.)/, $format) {
        $result .= $chunk =~ /^%\(.)/ ? strftime{$1}->($date,$lang) : $chunk;
    }
    return $result;
}

```

1.7 Generating Correct Charset Headers

When the page is produced it's important to specify a correct charset, so the browser will do the right thing when rendering the output. There are two techniques to accomplish that:

- The preferred method of indicating the encoding is by using the charset parameter of the Content-Type HTTP header. For example, to specify that an HTML document uses ISO-8859-1, a server would send the following header:

```
Content-Type: text/html; charset=ISO-8859-1
```

With mod_perl you can do that with:

```
my $r = shift;
$r->send_http_header('text/html; charset=ISO-8859-1');
```

- A less preferred method of setting the character encoding is by using the following tag in the HEAD section of an HTML document:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
```

This method requires that ASCII characters stand for themselves until after the <META> tag and often causes an annoying redraw with Netscape. The META HTTP-EQUIV method should only be used if one cannot set the charset parameter using the server.

Of course if you don't specify this header and a browser fails to guess the right character encoding, users may adjust it manually by telling the browser to re-display the page using a selected encoding. But this method is really against service usability: in many browsers the user will have to change the encoding again and again, as the browser will fall back to use its default value after each click.

1.8 Conclusions

We have discussed the following multi-lingual site development issues:

- It's almost always better to develop a dynamic site rather than a static one.
- Language selection is done by either asking the user about it and looking at the Accept-Language header.
- Memory of the user preference is best done via cookies, or by making the user log in.
- We have seen that the generated output is comprised from semi-static template text and dynamic database content.
- We have seen how site browsing is different from site searching in the terms of multi-lingual input processing and seen the code that handles that.
- We have discussed ways the language specific data can be stored in the database.
- We have seen how multilingual variants of text can coexist in the same template and have the code deal with that.
- We have seen how the presentation of dates and time can be adjusted to the user preferences.
- Finally we have learned how to tell client browsers to render the output using a correct language encoding.

For more information about discussed topics please refer to the *Resources* section.

In addition a simple introduction to mod_perl follows in your handouts.

1.9 Resources

- mod_perl home page: <http://perl.apache.org/> mod_perl guide: <http://perl.apache.org/guide/>
- Template Toolkit home page: <http://www.template-toolkit.org/>
- Internationalization / Localization: Charset parameter: <http://www.w3.org/International/O-HTTP-charset.html>

- “A tutorial on character code issues” by Jukka Korpela: <http://mirror.subotnik.net/jkorpela/chars.html>
- “Localizing Your Perl Programs” by Sean Burke and Jordan Lachler. (The Perl Journal issue 13 spring 99.)
- CPAN modules mentioned in this talk:

HTTP::Negotiate: <http://search.cpan.org/search?dist=libwww-perl>

HTML::Entities: <http://search.cpan.org/search?dist=libwww-perl>

Apache::Util: http://search.cpan.org/search?dist=mod_perl

Apache::Language: <http://search.cpan.org/search?dist=Apache-Language>

;o)

2 mod_perl: Getting Started Fast

2.1 mod_perl in Four Slides

Each tutorial will concentrate on different aspects of running a mod_perl server and mod_perl programming. In case you don't know how to get started with it, or you think it's a difficult task, these slides will take away any worries you might have had when you came to this tutorial.

In just four slides you will be able to install and configure a mod_perl server. And, of course, to write new code and reuse the existing code under mod_perl.

The four slides (sections) are:

- Installation
- Configuration
- The “mod_perl rules” Apache::Registry Scripts
- The “mod_perl rules” Apache Perl Module

2.2 What is mod_perl?

But before we go any further, there is a chance that you don't know what mod_perl is. So let's make a little introduction to mod_perl.

Everybody knows that Perl scripts running under mod_cgi have numerous shortcomings. There are many of them, but code recompilation and Perl interpreter loading overhead at each request is the hardest one to overcome.

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the <http://perl.apache.org/netcraft/> as of January 2001 about 2 million hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the Apache Software License.

mod_perl does away with mod_cgi's forking by reusing the existing child processes. In this new model, the child process doesn't exit anymore when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since the interpreter is persistent throughout the process' lifetime, all code is loaded and compiled only once, the first time it is seen. This makes all subsequent requests run much faster because everything is already loaded and compiled. Response processing is now reduced to running your code. This improves response times by a factor of 10 to 100, depending on the code being executed.

Doug didn't stop here, he went and extended mod_cgi's functionality by adding a complete Perl API to the Apache core. This makes it possible to write a complete Apache module in Perl, a feat that used to require coding in C. From then on mod_perl enabled the programmer to handle all phases of request processing in Perl.

The new Perl API also allows complete server configuration in Perl. This has which made the lives of many server administrators much easier, as they could now benefit from dynamically generating the configuration, freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.

To provide backwards compatibility for plain CGI scripts that used to be run under `mod_cgi`, while still benefiting from a preloaded perl and modules, a few special handlers were written, each allowing a different level of proximity to pure `mod_perl` functionality. Some take full advantage of `mod_perl`, while others only a partial one.

`mod_perl` embeds a copy of the Perl interpreter into the Apache `httpd` executable, providing complete access to Perl functionality within Apache. This enables a set of `mod_perl`-specific configuration directives, all of which start with the string `Perl*`. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) makes a very, very large program. `mod_perl` certainly makes `httpd` significantly bigger and you will need more RAM on your production server to be able to run many `mod_perl` processes, but in reality the situation is different. Since `mod_perl` processes requests much faster, the number of the processes needed to handle the same request rate is much lower relative to the `mod_cgi` approach. Generally you need slightly more memory available, and the speed improvements you will see are well worth every megabyte of memory you can add.

Now let's get back to the *All-In-Four-Slides...*

2.3 Installation

Did you know that it takes about 10 minutes to build and install a `mod_perl` enabled Apache server on a computer with a pretty average processor and a decent amount of system memory? It goes like this:

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That's all!

- Of course you must replace `x.x.x` with the actual version numbers of the `mod_perl` and Apache releases that you use.
- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

All that's left is to add a few configuration lines to a *httpd.conf*, an Apache configuration file, start the server and enjoy `mod_perl`.

2.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
# for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes every URI starting with */perl* to be handled by the Apache `mod_perl` module. It will use the handler from the Perl module `Apache::Registry`.

2.5 The "mod_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under `mod_cgi`:

```
mod_perl_rules1.pl
-----
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
-----
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

```
http://localhost/perl/mod_perl_rules1.pl
http://localhost/perl/mod_perl_rules2.pl
```

In both cases you will see on the following response:

```
mod_perl rules!
```

2.6 The "mod_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a handler subroutine and return the status to the server.

```
ModPerl/Rules.pm
-----
package ModPerl::Rules;
use Apache::Constants;

sub handler{
    my $r = shift;
    $r->send_http_header('text/plain');
    print "mod_perl rules!\n";
    return OK;
}
1;
```

Create a directory called *ModPerl* under one of the directories in @INC, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules
<Location /mod_perl_rules>
    SetHandler perl-script
    PerlHandler ModPerl::Rules
</Location>
```

Now you can issue a request to:

```
http://localhost/mod_perl_rules
```

and just as with our *mod_perl_rules.pl* scripts you will see:

```
mod_perl rules!
```

as the response.

2.7 Is That All I Need To Know About mod_perl?

Definitely not!

These slides are intended to show you that you can install and start using a mod_perl server within 30 minutes of downloading the sources.

There is much more to mod_perl than this, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

;o)

Table of Contents:

Tutorial: Multilingual sites with mod_perl and Template Toolkit	1
mod_perl Tutorial: Multi-lingual Sites with Perl	3
1 Multi-lingual Sites with Perl	3
1.1 Agenda	4
1.2 Dynamic versus Static Pages	4
1.2.1 User Language Detection	4
1.3 Generating the Content.	6
1.4 Fetching Dynamic Data	6
1.4.1 Searching	7
1.4.2 Browsing	9
1.4.3 Data Retrieval	9
1.5 Invariant Data	10
1.6 Handling Dates and Time Presentation	14
1.7 Generating Correct Charset Headers	18
1.8 Conclusions	19
1.9 Resources	19
mod_perl Tutorial: mod_perl: Getting Started Fast	21
2 mod_perl: Getting Started Fast	21
2.1 mod_perl in Four Slides	22
2.2 What is mod_perl?	22
2.3 Installation	23
2.4 Configuration	24
2.5 The "mod_perl rules" Apache::Registry Scripts	24
2.6 The "mod_perl rules" Apache Perl Module	25
2.7 Is That All I Need To Know About mod_perl?	25